Evaluation of XPath Queries on XML Streams with Networks of Early Nested Word Automata

Tom Sebastian

Université Lille 1 – Sciences et Technologies
Innovimax Sarl
Institut National de Recherche en Informatique et en Automatique
Centre de Recherche en Informatique, Signal et Automatique de Lille

Thèse

présentée en première version en vue d'obtenir le grade de Docteur par l'Université de Lille 1 en spécialité Informatique

par

Tom Sebastian

Evaluation of XPath Queries on XML Streams with Networks of Early Nested Word Automata

Thèse soutenue le 17/06/2016 devant le jury composé de :

Carlo Zaniolo	University of California	Rapporteur
Anca Muscholl	Université Bordeaux	Rapporteur
Kim Nguyen	Université Paris-Sud	Examinateur
Remi Gilleron	Université Lille 3	Président du Jury
Joachim Niehren	INRIA	Directeur

Abstract

The eXtended Markup Language (XML) provides a format for representing data trees, which is standardized by the W3C and largely used today for exchanging information between all kinds of computer programs. The main application areas of XML are document processing (DocBook), Web information representation (Html), and NoSql databases with nested relations. The main task for XML processing are the problems to query and transform data trees. For solving these tasks, the W3C has developed the languages Xslt, XQuery, and XProc. As a common core, they rely on XPath for querying data trees for nodes, strings, functions, or sequences thereof.

The challenge that we tackle in this thesis is the problem of how to answer XPath queries on XML streams with low latency, full coverage, high time efficiency, and low memory costs. This requires to overcome the following difficulties:

1. Earliest query answering is computationally untractable, so that one cannot hope to always reach optimal latency and memory consumption.

2. No previous streaming algorithm was able to deal with all navigational XPath queries while also supporting comparisons of data values, aggregates, and higher-order functions.

3. In contrast to in-memory XPath engines, where document projection with respect to a given XPath query is highly relevant for time efficiency, it was unclear how to project away irrelevant parts of XML streams.

In this thesis we first propose to approximate earliest query answering for navigational XPath queries by compilation to early nested word automata. It turns out that this yields quasi-optimal results in most practical cases, leading to almost optimal latency and memory consumption. Second, we contribute a formal semantics of XPath 3.0. It is obtained by mapping XPath to the new query language λXP that we introduce. This language has the advantage to be based on first principles, so that it can be considered as a core language for XPath processing. We then show how to compile λXP queries to networks of early nested word automata, and develop streaming algorithms for the latter. Thereby we obtain a streaming algorithm that indeed covers all of XPath 3.0. Third, we develop an algorithm for projecting XML streams with respect to the query defined by an early nested word automaton. Thereby we are able to make our streaming algorithms highly time efficient. We have implemented all our algorithms with the objective to obtain an industrially applicable streaming tool, and tested them on the usual benchmarks. It turns out that our algorithms outperform all previous approaches in time efficiency, coverage, and latency.

Acknowledgments

First of all, I want to express my gratitude to my doctoral advisor Joachim Niehren. You have introduced me to many different topics that we covered in my thesis and I appreciate all that I could learn. It has always been a pleasure to work with you! To me you have been more than just a great teacher and supervisor. Thank you for having been on my side with the various problems that we lived through during my thesis. Thank you that you cared so much for me. It was a great time. I will miss it!

I want to thank Carlo Zaniolo and Anca Muscholl for their great examination and review of this thesis. I like to thank Kim Nguyen and Remi Gilleron who took part in the jury of my defense, and Iovka Benova and Sławek Staworko for proof reading parts of my thesis. Thanks also to Mohamed Zergaoui, who always had trust in me and who financed this thesis.

I was gifted with a very pleasant work environment with handsome colleagues. I have been very fortunate with Denis, who implemented a great deal of the software tool that we have developed during my PhD. He was always helpful and I could enjoy many interesting discussions with him on code and algorithms. In addition, I especially appreciated the presence of Iovka, Antoine, Pavel, and Momar.

Huge thanks I dedicate to my family. Especially my lovely parents have supported me in so many ways throughout my thesis. I cannot thank you enough! Also I could not have wished for a better brother. Thanks Ron also for equipping me with the newest of technology. “Meine Mama” :)

I am so grateful for my best friends. Eric: thank you so much for being such a good friend for so many years, thanks for putting so much effort into our friendship, for the sports, the competitiveness, and for your kindness. Frank: “hey hey hey” :) thank you also for being such a great friend for all this time, especially for being so charismatic; you have the talent to put a smile onto anybody’s face :) Olaaaa: I “love” you :) I am so happy to have met you. You are the best! Bisousss. Sławek: I enjoy every time when we meet. Thanks for your marvelous hospitality and all the times that we spend together.

Eglantine, my little flower, thank you for being so sweet and to have come into my life and heart. “Eggy Spaghetti P***t” ;)

Last but not least, I want to acknowledge the great Snooker times with Mathias, the evenings with Gwenael and Anne-Sophie, the cooking by Guillaume and the Risotto of Michele, the dish washing with Wassim, and finally all great times that I shared with my flat mates Manon, Sandra, Vincent, and Alexandra.

Contents

Chapter 1

Introduction

Contents

The eXtended Markup Language (Xml) provides a format for general information representation [Bray 2008], which is standardized by the World Wide Web Consortium (W3C) and largely used today for exchanging information between all kinds of application. Essentially, the Xml format provides a concrete syntax for describing data trees, i.e., finite unranked trees whose nodes are labeled by data values.

There are three communities that joined their forces within the W3C for creating Xml that correspond to the following application areas: document processing as with DocBook [Walsh 2010a] or Sgml [Goldfarb 1986], Web information representation as with HTML [Pemberton 2002, Hickson 2014], and NoSql databases with nested relations [Meier 2000, Grün 1983, Bloom 2001], where NoSql stands

```
<twitter>
  <tweet><id>2</id><user>984</user><mess>@PPeter @Marctotheus My
    washing machine just broke down :(</mess></tweet>
  <tweet><id>4</id><user>349</user><reply>2</reply><mess>
    @IthinkThat I advise you to buy a Miele one. Those run forever
    </mess></tweet>
  <tweet><id>6</id><user>984</user><reply>4</reply><mess>@PPeter
    Thx, I will definitely check them out.</mess></tweet>
</twitter>
```

Figure 1.1: An XML document.

for "not only SQL" [Beyer 2011, Robinson 2015]. XML was a success story in the first two domains, with biggest success for the usage of input and output format for all kinds of computer programs, so that it can be avoided to write any ad-hoc parsers.

XML had fewer success with NoSQL databases and the development of Web services, for which a new format of data trees was developed, called the Java Script Object Notation (JSON) [Crockford 2001], which despite of its name is independent of the programming language of Web browsers Java Script [Eich 2015]. The data trees of JSON can be seen as nested key-value stores, i.e. as unranked unordered trees, in which each edge is annotated by a data value. Furthermore, a key constraint must be satisfied, stating that no two edges outgoing from the same node carry the same data value. The development of recent kinds of NoSQL databases relies on the JSON format rather than on XML, since focussing on distributed query answering algorithms [Friedgut 2004, Atserias 2013, Ngo 2013, Beame 2014, Beame 2013] in contrast to previous XML databases. The data trees underlying JSON and XML are not that different and can be converted to the other data tree, respectively.

1.1 Xml Processing based on XPath

The main task for XML processing are the problems to validate, query and transform data trees. An example is the data tree of Figure 1.2 that models a collection of tweets that reply to each other, i.e. it models the XML document in Figure 1.1.

1.1.1 Schema Validation

Validation of XML documents means to verify their conformance against a given schema. For instance, for the data tree of Figure 1.2 one would like to verify the existence of `mess`-child of a `tweet`-descendant that contains a non-empty data value, as with XPATH filter

```
[//tweet/mess[not(. = "")]].
```

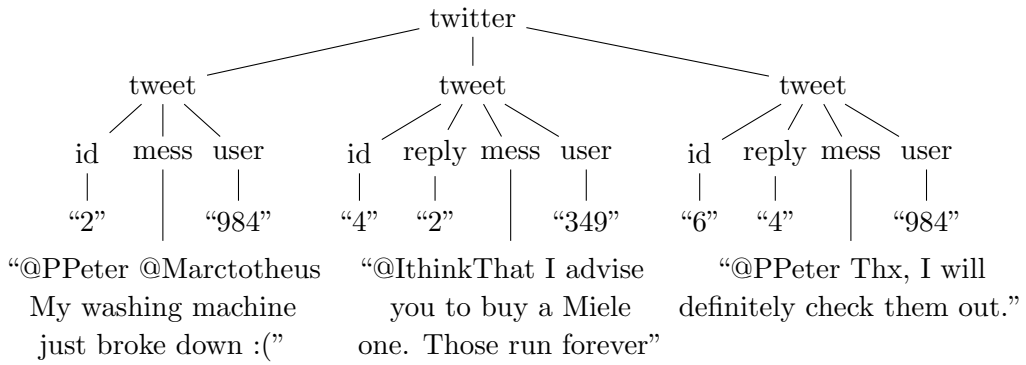


Figure 1.2: A data tree representing the XML document of Figure 1.1.

One would also want to check that every assigned data value of an `id` node of a tweet is unique. This can be done with the following XPATH query that uses function `distinct-values` that returns the input sequence of `id` values, but with duplicates removed:

```
count(//tweet/id) = count(distinct-values(//tweet/id/text()))
```

There are several standard schema languages performing validation. The W3C started with DTDS [Bray 2008] and then moved to more powerful XML Schema [Thompson 2004]. For instance, the syntax of XHTML is defined by a XML Schema, and also the syntax of DOCBOOK, or the syntax of MATHXML for representing mathematical formulas. The document community supported Relax NG [van der Vlist 2003], which was standardized by ISO. For instance, the syntax of the system biology markup language SBML is defined in Relax NG, while it was previously defined in XML Schema.

More recently, Schematron [Jelliffe 2006] was developed for an alternative validation approach, where schemas are expressed by XPATH filters. This has the advantage that errors can be localized more easily, so that error messages can be more informative. An example is the following XPATH filter that verifies that all `tweet`-labeled descendants have an `id`, a `mess`, and a `user`-labeled child. A corresponding Schematron program would output an error message for every missing `id`, `mess`, or `user`-child of a `tweet` descendant, and vice versa could report that a particular child exist.

```
[not(//tweet[not(id) or not(mess) or not(user)])]
```

The expressiveness of these schema languages was studied in numerous research papers [Martens 2006b, Martens 2006a]. It was shown that when ignoring data comparisons, DTDs describe vertically local languages of unranked trees, while XML schemas describe top-down and left-to right deterministic languages of unranked trees, so that they can be validated in streaming mode. The expressiveness

of Schematron, when restricted to XPath filters without advanced concepts (see Section 1.1.3) is restricted to first-order logic, while Relax NG supports general tree automata, and thus monadic second-order logic [Thatcher 1968, Comon 2007, Gottlob 2002].

1.1.2 Query-based Transformation

For solving the query-based transformation tasks, the W3C has developed the transformation languages Xslt [Clark 1999a], XQuery [Boag 2007], and XProc [Walsh 2010b]. As a common core, they rely on XPath [Clark 1999b] for querying data trees for nodes, strings, functions, or sequences thereof. In XQuery, transformations can then be expressed, similar to Sql data bases, by FLWOR expressions (`for`, `let`, `where`, `order by`, and `return`) as in XQuery

```
for $x in doc("twitter.xml")//tweet
where contains($x/mess,"washing machine")
order by $x/id
return $x/following-sibling::tweet[reply = $x/id]
```

which returns a list of all `tweet`-nodes that reply to some tweet whose message contains `"washing machine"`. The core of XQuery studied in [Benedikt 2009] exactly captures the class of first-order definable transformations.

Xslt transformations are expressed via stylesheets that declare a set of transformation rules that are based on XPath expressions. The core of Xslt can be defined by navigational macro tree transducers, where the navigation is defined by XPath queries [Maneth 2005]. For example the following Xslt program produces Html code that displays a table of all tweet ids and their messages.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html><body>
    <h2>Tweet Collection</h2>
    <table border="1"><th>ID</th><th>Message</th>
      <xsl:for-each select="twitter/tweet">
        <tr><td><xsl:value-of select="id" /></td>
        <td><xsl:value-of select="mess" /></td></tr>
      </xsl:for-each>
    </table>
  </body></html>
</xsl:template>
</xsl:stylesheet>
```

As shown by Maneth and Engelfriet [Engelfriet 2000], macro tree transducers with bounded copying capture the class of all MSO-definable transformations. This

remains true, when adding XPath queries for navigation, as long as these XPath queries remain first-order definable.

XProc is an Xml pipeline language whose programs allow to compose several processing tasks and transformations.

1.1.3 XPath

XPath [Clark 1999b] is a language standardized by the W3C for defining node selection queries in data trees. The selection conditions may rely on XPath filters that we encountered already in the section on Xml schemas.

XPath was designed to be a core language hosted by all other Xml processing languages of the W3C, and in particular for defining transformations of data trees based on node selection queries with Xslt, XQuery, and XProc. Currently, there are three W3C recommended versions of XPath [Clark 1999b, Berglund 2010, Robie 2014a] with version numbers 1.0, 2.0, and 3.0. The syntax got extended with each version in order to increase the expressiveness as we will discuss below, while the semantics basically remained unchanged except for the handling of errors and exceptions, and for changes in the type system. Furthermore, the current XPath 3.0 language is given *two* possible semantics, the *ordered* semantics used in Xslt transformations and standardized in the W3C document about XPath 3.0 and the *unordered* semantics that is used by XQuery as stated in the W3C document on XQuery 3.0 [Robie 2014c]. The difference between these two semantics is that sets are either treated as unordered collections or as sequences in the order of their elements.

This difference is often crucial when it comes to stream processing, since when imposing an order on sets, the latency and buffering needs may increase dramatically. For instance, if one wants to select all tweets with a reply on a twitter stream, but the first tweet does not have any reply, then one must wait until the last tweet arrives, before one can be sure that the first tweet is not selected. In an ordered mode, one needs to output the selected tweets in their order, so the output of all other answers has to wait on the decision for the first tweet, and thus until the very end.

The core of XPath is a navigational logical language that provides path queries for node selection in Xml data trees [Gottlob 2002, Gottlob 2003]. For instance, the following XPath query selects all message nodes of tweets, that do not have an empty data value:

```
//tweet/mess[not(. = "")].
```

Navigational queries may navigate in a data tree moving along the tree relations such as child, descendant, parent, ancestor, following sibling, preceding sibling, following, preceding, etc. They may also test whether the label or data value of a node is equal to some constant.

One of the motivations of the design of XPath was that it should support all first-order connectives, as usual with database queries. This should also hold for its navigational core which is a variable free language in the style of modal and

temporal logics [Arenas 2007]. We already encountered negation in the previous XPath example. Filters of a node express existential properties, stating that some node can be reached when navigating over some path, and multiple filters express conjunctions. Based on negation, universal properties can be expressed by existential properties, and disjunctions by conjunctions. Also existential and universal quantifications exist within the language. For instance, filter

```
some $a in //a satisfies $a/b
```

asks whether there exists an `a` descendant that has a `b` child.

Marx and de Rijke [Marx 2004b] showed that the filters of the navigational core of XPath 1.0 can be mapped to first-order logic formulas with at most 2 variables per subexpression. The variables that are counted here are not only the free variables – of which we need one for node selection and none for filters – but also the free variables of the subexpressions, which may be globally bound. It is well-known, however, that the two variable fragment is not complete for first-order logic of trees. A counter example is the query

$$\texttt{a/b}^{*}\texttt{/c}$$

which selects all c nodes x with an a-ancestor y such that all intermediate nodes z on the path from y to x are labeled by b. Here, three variables are needed, x, y, and z. Since using recursion based on the Kleene-star, this query is not supported by the syntax of XPath (in any version), and even worse, it cannot be expressed in the navigational core of XPath 1.0.

This incompleteness problem of XPath was solved starting with version 2.0 [Berglund 2010], which added the missing expressiveness by adding operator `except` for path complementation. There the missing query can be expressed as follows[1]:

```
/a//c except /a//*[not(fn:local-name(.) = "b")]//c.
```

As a consequence the navigational core of XPath became first-order complete, since 3 variables are known to be enough for expressing arbitrary first-order properties of trees [Marx 2004a].

Besides the navigation core, XPath also permits to select, compare, and aggregate data values in data trees, and to apply all kinds of functions to them, including the usual arithmetic functions and string operations. For first-order logics of data trees with comparisons of data values, however, the 3 variable fragment is no more sufficient. Therefore XPath 2.0 did also add variables to the language. Query

```
for $t in //tweet return following-sibling::tweet[reply = $t/id],
```

for example, returns all tweets that reply to some start tweet, i.e. with the XQuery unordered semantics. It binds to variable `$t` progressively all `tweet` descendants, and returns all following-sibling `tweet` nodes whose `reply` child is equal to the `id`

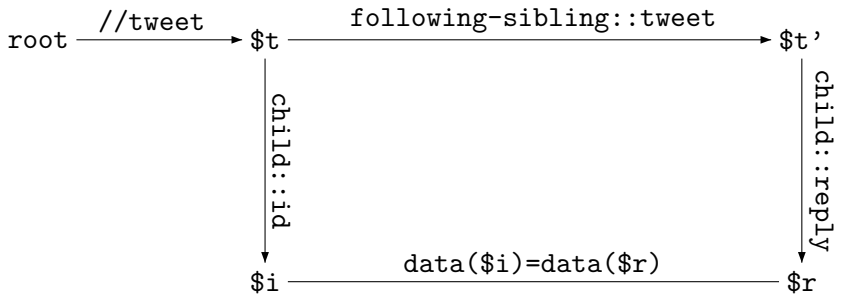


Figure 1.3: Graph of first-order query for XPath query `for $t in //tweet return following-sibling::tweet[reply = $t/id]`.

child of the node at `$t`. A corresponding first-order formula requires at least 4 variables as in query $\{\texttt{\$t'} \mid \exists \texttt{\$t}, \texttt{\$i}, \texttt{\$r} : \Phi(\texttt{\$t'}, \texttt{\$t}, \texttt{\$i}, \texttt{\$r}), \Phi$ presented in Figure 1.3$\}$.

XPath 3.0 comprises a new type system that contains a large domain of data types that are based on sequences. Sequences are ordered shallow lists, that are not nested, whose single element instances are identified with the element itself, making XPath 3.0 an expression language for processing sequences. Query

```
for $x in (1, 2, 3) return $x + 4
```

returns sequence `(5, 6, 7)`, for example. The conversion of data types is also supported within XPath, for example, it provides conversion function `fn:boolean` that converts numbers, strings, and sequences of nodes to a boolean, allowing for richer test expressions for conditionals and filters, as in query

```
if ("abc") then fn:true() else fn:false()
```

that returns *true*, since `fn:boolean("abc")` = *true*. Furthermore, XPath can also express aggregation queries and supports higher-order functions, such that even first-order logic is not enough!

Functions have become first-class citizens in XPath 3.0 and they may be of higher order. They can be defined inline within expressions as in query

```
function($a as xs:double, $b as xs:double) as xs:double {$a * $b}
```

that returns the product function, or functions can be returned by partial function applications, such as

```
fn:concat("a", ?, "c")
```

which returns the anonymous function $\lambda y.$`fn:concat("a",` y`, "c")` that accepts one argument.

[1] XPath function `fn:local-name` returns the label of the current node. Note that filter `[not(b)]` is short for `[not(child::b)]` and thus not a valid substitution.

1.1.4 Regular Extension of XPath

Extensions of XPath by a Kleene-star operator for recursive path composition were proposed by many research papers [ten Cate 2007, Groz 2012, Parys 2011]. Recursive path queries are already interesting for navigational XPath queries such as `a/b*/c`. These can be expressed otherwise by using path complementation (as available since XPath 2.0). But this kind of encoding does not work in general. For instance one cannot express `a/(b/b)*/c` in XPath. Furthermore, the encoding of *conditional regular axis*, which have the form `b[F]*` where `F` is some filter, by using complementation is cumbersome, and spoils the combined linear time complexity for query answering.

In XPath 3.0 the Kleene-star operator is not supported syntactically, but it can be expressed in the general case by using recursive functions as we will explain later on. Nevertheless we will consider in this thesis an extension of XPath 3.0 with the Kleene-star operator that we call *regular* XPath *3.0*. The addition of the Kleene-star operator will mainly serve us for simplifying the presentation of recursive path queries in examples, and can be eliminated without problems by using recursive functions.

Recursive path queries are particularly relevant when combined with variables and comparisons of data values (data joins). For instance, we can use a regular XPath 3.0 query to select dialogs in a Twitter stream:

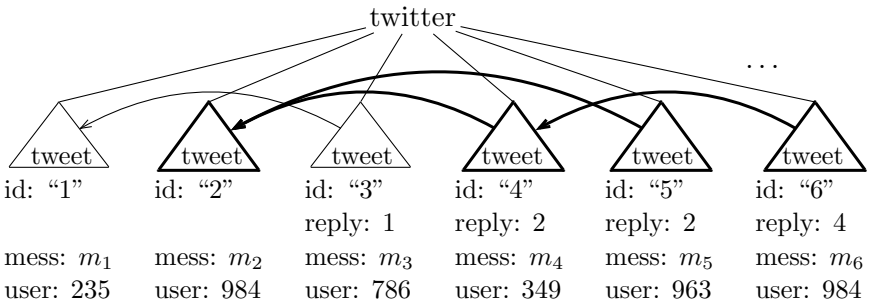
```
//tweet[fn:contains(mess, "washing machine")]
    /(let $y := .  return following-sibling::tweet[reply = $y/id])*
```

The query navigates to `tweet`-labeled nodes whose message `mess` contains "`washing machine`", and from there it navigates recursively to following-sibling tweets that reply to it. It thereby select the nodes of Figure 1.4 that are marked in bold. For the second tweet with id "2", the dialog query reports the tweet messages of repeated replies, finding tweets with ids "4", "5", and "6".

1.1.5 In Memory Evaluation

Evaluation algorithms for XPath queries receive an XPath query Q and an Xml document D as an input, and compute the set of query answers $Q(D)$. Two kinds of evaluators should be distinguished: those who store the whole Xml document as a data tree in main memory in a preprocessing phase where all kinds of indexes can be computed, and those which read the Xml document as a stream in an incremental left-to-right once-only manner.

The complexity of in-memory evaluation algorithms for XPath fragments found much interest in database theory. For navigational XPath queries without variables from the core of XPath, there exist evaluation algorithms that require combined linear time in the worst case, i.e. time in $O(|Q| \cdot |D|)$ [Gottlob 2005]. When adding comparisons of data values, query answering remains in polynomial time $O(|Q|^3 \cdot |D|)$, see Theorem 1.1 in [Bojańczyk 2011], which is even linear in the size of the



m_1 = I bought me a new pair of asics. And off I go.
m_2 = @PPeter @Marctotheus My washing machine just broke down :(
m_3 = @loosingg That's awesome ;)
m_4 = @IthinkThat I advise you to buy a Miele one. Those run forever
m_5 = @IthinkThat Oh, but it's not like I can do anything about it.
m_6 = @PPeter Thx, I will definitely check them out.

Figure 1.4: A Twitter stream containing a dialog.

database. When recursive paths are also permitted, the worst-case complexity increases slightly to $O(|Q|^3 \cdot |D| \cdot log|D|)$. In contrast to these theoretical algorithms, the algorithm from Saxon may run in exponential time in the size of the database even for navigational queries. The problem may arise for queries where the number of alternations between forwards and backwards axis is not bounded, as illustrated by example in [Parys 2011].

The weakness of all these theoretical algorithms is that they traverse the whole database D even for queries Q that require to inspect only the local environment of a given tree node. This important problem can be approached in some algorithms by adding data tree projection as proposed in [Marian 2003, Maneth 2010]. The idea is to discover irrelevant parts of the database during query answering, and to avoid inspecting them as if they were pruned away. Indeed, the projection algorithm for XQuery from [Marian 2003] is used in Saxon [Kay 2004], the most used Xml processing tool for XQuery and Xslt as of today. Good projection algorithms for Xslt are missing though. While not available in Saxon, there exist good projection algorithms for large navigational fragments of XPath [Maneth 2010, Arroyuelo 2015] based on tree automata techniques with highly efficient implementations based on clever indexing.

1.1.6 Streaming Evaluation

When one wants to select dialogs in Twitter streams in real time, one cannot wait until the end of the stream arrives. Instead one would like to query it as stream of complex events – having a nesting structure – in real time and in an incremental manner. This requires to buffer all tweets of the dialog, since replies to any of these tweets may arrive with arbitrary delay. This is in contrast to sliding window techniques for monitoring continuous streams, which will buffer only a bounded

number of subsequent events on the stream [Barbieri 2010, Phuoc 2011].

The task of online dialog selection in Twitter streams is a special case of complex event processing (CEP) [Brenna 2009, Wu 2006, Mozafari 2012], which is equally relevant for the analysis of stock markets, RSS feeds, and social network posts. In the case of dialog selection, CEP can be reduced to querying XML streams, i.e., linearizations of data trees in XML format. As argued by [Mozafari 2012], this also holds for the other CEP tasks mentioned above.

The problem to answer XPATH queries on XML streams has been studied for more than a decade [Gupta 2003, Fernández 2007, Olteanu 2007, Schmidt 2007, Benedikt 2007, Madhusudan 2009, Kay 2010, Gauwin 2011a, Mozafari 2012]. The main motivation here was to deal with huge documents which do not fit into main memory, or large documents (say 40 gigabytes) for which the time to create an in-memory representation would impose a too long delay.

Low Latency. Suppose that we want to select all tweets that have a reply on a Twitter stream, while we assume the unordered mode for the output set. The natural idea to do so works as follows. Any new tweet that arrives on the stream is an *alive* candidate, of which one does not know yet whether it will become an answer. Therefore it must be buffered until the decision can be taken. Furthermore, whenever a newly arriving tweet is a reply to a previous one in the buffer, then this buffered tweet is a candidate that can be safely selected. In this case it can be output and removed from the buffer.

More generally, the idea for answering queries on XML streams is to buffer all *alive* candidates, until one can decide whether they answer the query or not for all continuations of the stream. An algorithm that buffers only alive candidates is called *earliest*. Any algorithm performing *earliest query answering* (EQA) must output query answers with lowest latency and discard failed candidates at the earliest time point. Unfortunately, it turns out that EQA is not feasible in polynomial time for XPATH queries [Benedikt 2008b], as first shown by adapting counter examples from online verification [Kupferman 2001]. A second hardness argument follows from that deciding aliveness is more difficult than XPATH satisfiability [Gauwin 2011a], which is coNP-hard even for small fragments of XPATH [Benedikt 2008a].

The situation is different for queries defined by deterministic *nested word automata* (NWAS) [Alur 2004, Alur 2009], for which earliest query answering is feasible in cubic time in the size of the automaton [Madhusudan 2009, Gauwin 2009b]. This can be applied to navigational XPATH queries by compilation to NWAS [Gauwin 2011a], but this does not lead to an efficient streaming algorithm. The first problem is that cubic time per event is too much in practice. The second problem is that the compiler from XPATH to NWAS relies on nondeterminism for descendant and following axes, and that the determinization of NWAS often raises huge blow-ups in practice (in contrast to finite automata).

Since EQA is not feasible, existing streaming algorithms for XPATH evaluation either approximate EQA or restrict the problem to 0-delay queries. Approximations

are done by SPEX's, which compiles navigational XPATH queries to transducer networks [Olteanu 2007], but also by SAXON's streaming XSLT engine [Kay 2010], and GCX [Schmidt 2007] which implements a fragment of XQUERY.

The recent XSEQ tool [Mozafari 2012], in contrast, restricts XPATH queries by ruling out complex filters all over. In this way, node selection can always be decided with 0-delay [Gauwin 2011b] once having read the attributes of the node (which follow its opening event). Such queries are called begin-tag determined [Benedikt 2007] if not relying on attributes.

Low Memory. A principle question is how much memory one may want to allow for the streaming processing. Given that a stream can be read only once, from left to right, one has to buffer parts of the stream that will be needed later on again. The main objective may be to minimize the size of the buffer as with EQA, given that streams may be huge or even infinite (in the case of TWITTER).

Ideally, one would like to grant only constant memory, and thus be independent of the size of the stream. But this allows hardly any query to be evaluated in streaming mode. First of all, streaming algorithms for data trees typically have a stack whose depth is equal to the depth of the tree, so if the depth of the tree is not bounded, the memory will not be bounded. In practice, however, the depth of the tree may become large but rarely huge, so that the stack fits into the main memory. The second problem is about the size of the buffer which may grow linearly with the size of the document, for instance for the query that selects all tweets with a reply on a Twitter stream, where no tweet has a reply.

Besides the dependence on the depth of the XML tree and the number of candidates that have to be stored, many interesting queries require to buffer data content of the XML document. For instance, if one wants to verify that no two tweets have the same Id, one needs to store the Ids of all tweets. If the Twitter stream is large but not huge then it may as well happen that the set of all Ids fits into main memory while the whole stream does not.

Time efficiency. Time efficiency is of paramount importance for XPATH query answering on XML streams in practice. When the query is fixed, only constant time should be needed for each event, so that the overall time is linear in size of the stream. The typical case where this may fail even for navigational queries without variables, is when the number of buffered candidates grows with the size of the stream. In this case, one must avoid inspecting all the candidates in the buffer at any event.

When measuring the time efficiency, it is important to separate the pure parsing time from the time for answering queries. The reason is that one usually has to answer a collection of XPATH queries – stemming from an XSLT or XQUERY program – at the same time. The parsing time may be shared between all these XPATH queries while the time for answering the query cannot be shared. Furthermore, the parsing-free time for answering a single XPATH query is often dominated by the

parsing time, so that one may quickly take wrong conclusions when not separating these two time measures.

It is also interesting to compare the time requirements of streaming and in-memory algorithms. Both algorithms have to parse the XML document. While in-memory algorithms may compute some indexes during preprocessing, this cannot be done for streaming algorithms. Besides, the parsing-free time of a streaming algorithm should correspond to the time needed by an in-memory algorithm after preprocessing, except for the benefits of using indexes. The advantage of streaming algorithms may be a lower latency. In applications such as streaming validation, this may help to avoid reading the whole document, and thus spare parts of the parsing time and of the preprocessing time. Generally, when only a few queries have to be answered on the same document, streaming evaluators tend to be more efficient than in-memory evaluators, since the cost of computing the indexes does often not get amortized when answering few queries only.

1.2 Open Problems

From the practical perspective the main open problem is that there is no streaming algorithm for XPath 3.0 with high coverage, low latency, high time efficiency, and low memory consumption. The question is whether such an algorithm may exist, given that EQA is computationally hard.

1.2.1 Low Coverage

The coverage of existing tools for navigational XPath is quite limited and the support of non-navigational concepts is even worse, in particular the support for variables, comparisons of data values, sequences, arithmetics, string operations, aggregations, and higher-order functions.

For instance, the GCX system is the only system that allows to evaluate XPath queries in streaming that perform some data comparisons, but in order to obtain this, the supported navigational aspects are minimalized. The navigational aspects are also strongly restricted in XSEQ, given that only 0-delay queries are supported, where the selection of a node can be decided after having read its opening tag and its attributes.

When looking into the usual XPathMark benchmark [Franceschet 2005] the best coverage is reached by Olteanu's SPEX [Olteanu 2007] with 22% of the use cases. The time efficiency of SPEX, however, is only average, for instance compared to GCX [Schmidt 2007] which covers 19% and which often runs in time close to the parsing time for a single query. Given that SAXON is mostly interested in XSLT and XQUERY evaluation, the support for the navigational aspects XPath remains limited, leading to 6% of coverage for the XPathMark benchmark.

1.2.2 Low Latency

Since EQA is computationally hard, one cannot hope for algorithms with minimal latency. For this reason, some approaches including the XSEQ system are restricted to 0-delay queries, in order to avoid buffer management all over [Mozafari 2012, Benedikt 2008b]. Other approaches such as SPEX do not impose delay restrictions, but have only low coverage since not supporting data joins, aggregation, higher-order functions, and recursion.

A good approximation of EQA is also essential to ensure low memory consumption. For this it is necessary to decide at a close to optimal time point whether a candidate is safe for selection or rejection for a given query. The earlier candidates can be selected or rejected, the lower will be the latency and memory consumption.

1.2.3 Low Time Efficiency

There are numerous reasons that limit the time efficiency of the existing streaming algorithms, many of which depend on the algorithmic approach that is chosen. One typical problem is to deal with queries with variables for tuple selection. In this case, the number of candidate tuples may even grow exponentially in the arity of the tuples. In the case of monadic node selection queries, the number of candidates is at most linear in the size of the stream. This may still raise time costs if all candidates are re-inspected at any event, typically for deciding whether they are still alive or already safe for selection or rejection. If the number of candidates grows linearly in the size of the stream, the overall running time will become quadratic, which is not feasible for large streams.

Another general problem is the lack of projection algorithms for streaming XPATH query evaluation. This is an important problem, given that projection is essential for the performance of in-memory evaluators. So the question is how to obtain good projection algorithms for stream processing, by which to reduce the parsing-free query answering time.

1.2.4 Lack of Formal Semantics.

Similarly to all other W3C standards XPATH has no formal semantics. The problem is that the standardization documents contain only informal descriptions of some hundreds of pages often specified on an abstraction level close to implementation. For example the XPATH 3.0 grammar [Robie 2014b] already contains more than one hundred nonterminal symbols and there are hundreds of functions and operators [Kay 2014]. As a result, those definitions cannot serve as a basis for a formal analysis and moreover they are difficult to understand.

So the challenge is whether one cannot have a formal core language of XPATH that is based on first principles, to which all the different versions of XPATH can be mapped by following their specification in the standards. Defining such a core language seems also unavoidable if one wants to obtain large coverage streaming implementations.

1.3 Contributions

Our main contributions is a collection of algorithms that allow to evaluate full XPath 3.0 queries on Xml streams with low latency and with high time efficiency in practice.

1.3.1 Formal Semantics of XPath 3.0 in λXP

We introduce λXP, a typed first-order logical language for querying data trees based on first principles, supporting path navigation, higher-order functions, recursion, aggregates, tuples, lists, strings, and numbers.

To the best of our knowledge, the language λXP is different to previous higher-order query languages [Benedikt 2015], in that λXP relies on a mode systems which permits to distinguish logical subformulas for which solutions must be searched, and functional subformulas that can be evaluated deterministically such as functional programs.

The purpose of λXP in this thesis is to serve as a core language for implementing evaluators for XPath 3.0. Therefore, we show how to compile all of XPath 3.0 to λXP with one minor exception. The compiler can be understood as a compact formalization of what is described informally in the specification of XPath 3.0. Even though formal, we believe that some of the details of the XPath 3.0 specification by the W3C are easier to understand through our compiler to λXP.

1.3.2 Early Query Answering by Early Nwas

Queries in the navigational fragment of λXP can be compiled to nested work automata (Nwas) as shown in [Gauwin 2009b], under the assumption that there are no backwards axis.

Nwas are the natural kind of pushdown machines for processing nested words such as Xml streams [Alur 2004]. They can be determinized in contrast to standard pushdown automata, basically since they have the same expressiveness as tree automata. Furthermore, eqa for queries defined by deterministic Nwas can be done in cubic time per event [Gauwin 2009b, Madhusudan 2009]. This is an interesting result, but since the resulting algorithm always requires cubic time in practice, this is by far too slow. Furthermore, determinism is assumed, which must be obtained by static determinization, which may lead to an exponential explosion – which does arise for many practically relevant queries – and even before the cubic time algorithm is applied.

We propose an approximation of eqa for navigational λXP queries, which does not require static determinization. For this we introduce so called early Nwas, which besides final states have selection and rejection states. Whenever a selection (rejection) state is reached, any continuation of the nested word will be accepted (rejected). We then show how to introduce appropriate selection and rejection states when compiling navigational λXP queries to early Nwas. What we obtain is an approximation of eqa, since it may still be the case that some candidate is safe for

selection even though the early NWA is not in a selection state. But deciding whether an early NWA is in a selection state is much easier that deciding the aliveness of candidates, since the latter depends on its stack and its state. Indeed, the approximation is indeed exact for all positive navigational XPATH queries without valid or unsatisfiable subfilters as shown in [Lick 2013].

Our streaming algorithm for navigational λXP queries runs the corresponding early NWA for all possible candidates, while performing on-the-fly determinization. A candidate is created and buffered only if it does not go into a selection or rejection state. A candidate from the buffer is output when the state of the candidate is a selection state, and discarded when it is a selection or rejection state.

In order to avoid inspecting all candidates at all events, we propose a stack and state sharing algorithm. The idea is that we can share the computation of any two candidates in the same state, so that the running time does not depend on the number of buffered candidates, but only on the number of states of the deterministic NWA created by on-the-fly determinization. Our streaming algorithm with stack-and-state sharing for answering queries by early NWAs is original and nontrivial, and enables tight upper bounds for time and space complexity that we prove.

1.3.3 Networks of Early Nwas

We present a streaming evaluator for all λXP queries and thereby for all of XPATH 3.0. The basic idea is that a formula of λXP can be decomposed into a network of navigational λXP formulas with only forwards axis, each of which can be compiled to an early NWA. Therefore, it remains to show how to evaluate networks of early NWAs in a streaming manner.

1.3.4 Projection for Nwas

We initiate the development of projection algorithms for processing XML streams, and in particular for NWAs.

Our idea is that a projected nested word should contain jump symbols $.^{i}.$ for projected factors, where the integer i stands for the excess of the factor, i.e., the difference between the number of opening and closing tags. We propose *projection nested word automata* (PNWAS), a kind of mixed pushdown and counting automata, that input projected nested words which besides others contain integers as letters. Conversely, a projection of a nested word with respect to a given NWA can be computed by any corresponding PNWA. We then lift NWA projection to the evaluation of navigational XPATH queries on XML streams. It turns out that the parsing-free time for our query answering algorithm for navigational XPATH queries reduces by a factor of 4 on average on the XPATHMARK benchmark.

1.3.5 Implementation and Experiments

We have implemented our algorithms for navigational XPath queries and tested and compared them using the usual XPathMark benchmark. Considering only the navigational fragment of XPath, we improve on all other tools in coverage with 37% of the XPathMark benchmark (the previous best is Spex with 22%). We also outperform all of them in time efficiency with the exception of Gcx, which runs slightly quicker on few queries, and slightly slower on others.

1.4 Organization

The thesis starts with Chapter 2 on preliminaries that discusses data trees and their linearization into nested words. It continues on Xml data trees that model the tree-structure of the Xml data model, and it finishes with Nested Word Automata, that allow to process linearizations of such data trees. This thesis is organized into three main parts.

Part one on "A Formal Semantics of XPath 3.0" describes how the semantics of XPath can be described formally by λXP expressions. It is split into the following five chapters. Chapter 3 on Navigational XPath describes the navigational XPath 3.0 fragment, Chapter 4 explains the types and values of XPath 3.0, while Chapter 5 presents full XPath 3.0. Various XPath expressions are explained abstractly, while not going into details of the XPath specification. Next, in Chapter 6 we introduce the λXP language that allows to query data trees and supports higher-order functions and recursion. Then we present a compiler from XPath to λXP in Chapter 7.

Part two on "Early Query Answering for Navigational XPath" describes how XPath can be evaluated on Xml streams using Early Nested Word Automata, and how it is speed up by introducing projection thereof. Chapter 8 on Early Nested Word Automata compiles a forward navigational fragment of λXP to early nested word automata which are run on Xml streams. We present a stack-and-state sharing algorithm that reduces the processing time per event to be linear in the number of states and stack symbols of the early nested word automata and we present and compare the efficiency of our streaming tool that implements the presented algorithms. In Chapter 9 we develop projection algorithms for nested word automata. For this we compile nested word automata into projection nested word automata that we introduce. These are used to produce the projected nested word on which it runs. We finish by an experimental section that shows the speed-up that we gain with respect to the early query answering algorithm of Chapter 8.

Part three on "Query Evaluation Algorithms for Full XPath" presents algorithms for evaluating λXP formulas on data trees in-memory and in streaming. Chapter 10 presents in-memory evaluators of λXP queries that rely on the mode system of λXP, which splits the logical and programming parts of λXP queries. Chapter 11 turns the in-memory evaluator into a streaming evaluation algorithm. It runs early nested word automata for navigational λXP queries, while decomposing

general λXP queries into a network of navigational queries.

We have added the following two appendices. Appendix A lists the full query collection of the XPathMark benchmark, which we use in our experiments. Appendix B gives a complete translation of the compiler from XPath 3.0 to λXP expressions that follows the ordered semantics of Xslt.

1.5 Publications

A quick sketch of our results of Early Nested Word Automata for XPath Query Answering on Xml streams in Chapter 8 was published in the CIAA'2013 conference [Debarbieux 2013]. We published these results with full proofs and extended experiments in the Theoretical Computer Science Journal [Debarbieux 2015]. Our projection algorithms in Chapter 9 were published in the SOFSEM'2016 conference [Sebastian 2016].

Chapter 2

Preliminaries

Contents

We introduce basic notations needed throughout the chapters. We start by data trees and nested words. Then we discuss how the XPath data model can be modeled using data trees, and show what Xml streams are. Finally we introduce Nested Word Automata.

2.1 Data Trees

Data trees in this thesis will be unranked ordered finite rooted trees in which any node is labeled by a tuple of data values. In addition, any node will also be labeled by a tag from a finite alphabet.

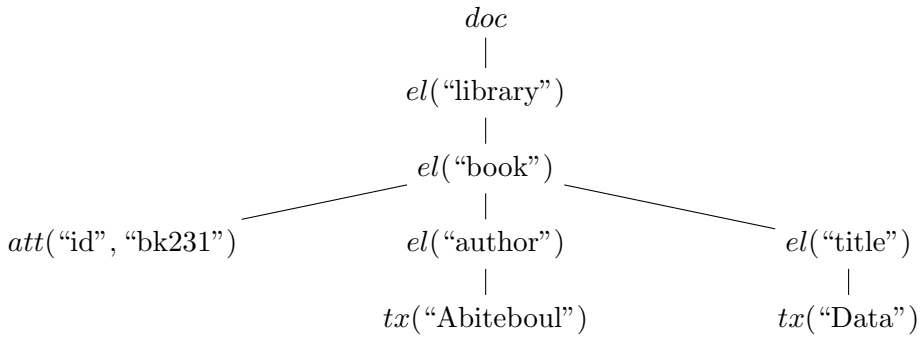


Figure 2.1: A sample of a data tree: $t_{2.1} \in \mathcal{T}_{\text{XML}'_{types},\text{UTF8}^*}$.

2.1.1 Definition

We fix a finite ranked alphabet Σ with arity function $ar : \Sigma \to \mathbb{N}$. Elements in Σ are called *node tags* and ranged over by σ. We will sometimes write $\sigma^{(k)}$ to denote that $ar(\sigma) = k$ for some $\sigma \in \Sigma$. Furthermore we let Δ be a finite set, whose elements will be called *characters of data values* or *internal letters* and ranged over by c.

We next define data trees with parameters Δ and Σ. A *data value* is a string over Δ, and ranged over by w, i.e., $w \in \Delta^*$. The set of node labels $\mathcal{A}$ is defined as follows:

$$\mathcal{A} = \bigcup_{\sigma^{(k)} \in \Sigma} \{\sigma\} \times \underbrace{\Delta^* \times \ldots \times \Delta^*}_{k}$$

The elements of $\mathcal{A}$ will be ranged over by a. The set $\mathcal{T}$ of *data trees* t over Σ and Δ is the least set that contains all tuples $a(t_1, \ldots, t_n)$ where $a \in \mathcal{A}$ and $t_1, \ldots, t_n \in \mathcal{T}$. In other words, the set of data trees satisfies the following abstract syntax:

$$t \in \mathcal{T} ::= a(t_1, \ldots, t_n) \quad \text{where } a \in \mathcal{A},\ n \geq 0, \text{ and } t_1, \ldots, t_n \in \mathcal{T}$$

We will sometimes write $\mathcal{T}_{\Sigma,\Delta}$ instead of $\mathcal{T}$ when we want to specify the signatures explicitly.

A data tree can be drawn as a graph in the obvious manner. An example motivated from the XML data model is the data tree $t_{2.1}$ given in Figure 2.1. Here, $\Delta = \text{UTF8}$ is the UTF-8 character set and $\Sigma = \text{XML}'_{types} = \{doc^{(0)}, el^{(1)}, att^{(2)}, tx^{(1)}\}$ contains some of the XML types. For element nodes the data value contains the XML tag, for text nodes the text content, and for attribute nodes the attribute name and its attribute string-value.

2.1.2 Logical Structure

Let $k = \max(\{ar(\sigma) \mid \sigma \in \Sigma\})$. For data trees, we define a logical structure with the following relational symbols:

$$\{ch, ch^+, ns, ns^+, par, par^+, ps, ps^+, fo, pc, fut\} \cup$$
$$\{tag_\sigma \mid \sigma \in \Sigma\} \cup \{dv_i \mid 1 \leq i \leq k\} \cup$$
$$\{equals\}$$

The domain of the structure of a data tree t will be the union of the set of the nodes of its graph together with the infinitely many data values in Δ^*. Here we consider the graph abstractly, in that nodes are identified by their Dewey address, i.e., by the word of natural numbers that addresses them when starting at the root. The root node of t is identified with the empty word ε, and the i-th child of a node π of t with $\pi \cdot i$.

More formally, the set of nodes of a data tree is defined recursively for all trees $a(t_1, \ldots, t_n)$ by:

$$Nodes(a(t_1, \ldots, t_n)) = \{\varepsilon\} \cup \{i \cdot \pi \mid 1 \leq i \leq n,\ \pi \in Nodes(t_i)\}$$

Another possible identification of nodes would be to simply enumerate them by a consecutive numbering in pre-order for example. The relation symbols are interpreted as binary relations on this domain as follows: The binary relation ch^t relates a node to its children, the binary relation ns^t relates a node to its next sibling to the right.

$$ch^t = \{(\pi, \pi \cdot i) \mid \pi \cdot i \in Nodes(t)\}$$
$$ns^t = \{(\pi \cdot i, \pi \cdot i{+}1) \mid \pi \cdot i,\ \pi \cdot i{+}1 \in Nodes(t)\}$$

The parent relation $par^t = (ch^t)^{-1}$ relates a node to its parent, and the previous sibling relation $ps^t = (ns^t)^{-1}$ relates a node to its previous sibling. Descendant relation $(ch^+)^t$, following-sibling relation $(ns^+)^t$, ancestor relation $(par^+)^t$, and the preceding-sibling relation $(ps^+)^t$ are the transitive closure of the child, next-sibling, parent, and previous-sibling relation, respectively: $(ch^+)^t = (ch^t)^+$, $(ns^+)^t = (ns^t)^+$, $(par^+)^t = (par^t)^+ = ((ch^+)^t)^{-1}$, and $(ps^+)^t = (ps^t)^+ = ((ns^+)^t)^{-1}$. For any relation A, by A^* we denote the reflexive closure of A^+. The following relation $fo^t = (par^t)^* \circ (ns^t)^+ \circ (ch^t)^*$ relates a node to all its following nodes, the preceding relation $pc^t = (par^t)^* \circ (ps^t)^+ \circ (ch^t)^*$ relates a node to its preceding nodes, and last but not least the future relation $fut^t = (ch^t)^* \vee fo^t$ relates a node to all its future nodes.

The label of a node $\pi \in Nodes(t)$ is denoted by $lab_t(\pi) \in \mathcal{A}$; it can be defined recursively for all $t = a(t_1, \ldots, t_n)$ and $i\pi \in Nodes(t)$ as follows:

$$lab_t(\varepsilon) = a$$
$$lab_t(i\pi) = lab_{t_i}(\pi)$$

tag_σ are unary relations that are interpreted as the set tag^t_σ of all nodes that are tagged by σ. For all $1 \leq i \leq k$ the functions dv^t_i map a node to the i-th data value of its label, i.e. $dv^t_i(\pi) = w_i$ whenever $lab^t(\pi) = \sigma^{(m)}(w_1, \ldots, w_m)$.

In order to test equalities among data values, we add function $equals^t$ that maps any two data values w_1 and w_2 to the boolean of whether w_1 is equal to w_2, i.e. $equals^t(w_1, w_2) = (w_1 = w_2)$. This operator is what we minimally wanted to have. It allows already to express the XPath join filter `[author[child::text() = "Abiteboul"]]` in the first-order logic of this relational structure, assuming that dv_1 for element nodes returns the Xml tag, and assuming that dv_2 for text nodes returns its data content:

$$\exists r.(\neg\exists.ch(y,r) \wedge tag_{el}(r) \wedge equals(dv_1(r), author) \wedge \\ \exists x.(ch(r,x) \wedge tag_{tx}(x) \wedge equals(dv_2(x), Abiteboul)))$$

But indeed our structure is incomplete in that its domain does not contain numerical values, nor functions, and in that no arithmetic operations and others are supported.

2.1.3 Finite State Abstractions

For applying finite state technology, it is often necessary to abstract information about data values, of which there are infinitely many, into finite state information. For instance, the set of Xml tags is infinite, but it can be abstracted into a finite set for a given query, for which only finitely many Xml tags are relevant.

More generally, we often want to relabel data trees. This can be done when given a relabeling function $\alpha : \mathcal{A} \to \mathcal{A}'$ for some other alphabet $\mathcal{A}' = \bigcup_{\sigma^{(k)} \in \Sigma'} \sigma \times (\Delta'^*)^k$. The relabeling of a data tree over Σ and Δ is then defined, such that for all $a(t_1, \ldots, t_n) \in \mathcal{T}_{\mathcal{A}}$:

$$rel_\alpha(t) = \alpha(a)(rel_\alpha(t_1), \ldots, rel_\alpha(t_n))$$

The relabeled data tree $rel_\alpha(t)$ then belongs to $\mathcal{T}_{\mathcal{A}'}$. For instance, if one wants to query the data tree in $t_{2.1} \in \mathcal{T}_{\text{Xml}'_{types},\text{Utf8}}$ in Figure 2.1 by a query that concerns only *author*-elements and data content, then we can use the following relabeling function α for abstracting the other information away.

$$\alpha(\sigma^{(k)}(w_1, \ldots, w_k)) = \begin{cases} author^{(0)} & \text{if } \sigma = el^{(1)} \text{ and } w_1 = author \\ tx^{(1)}(w_1) & \text{if } \sigma = tx^{(1)} \\ doc^{(0)} & \text{if } \sigma = doc^{(0)} \\ att^{(0)} & \text{if } \sigma = att^{(2)} \\ el^{(0)} & \text{if } \sigma = el^{(1)} \text{ and } w_1 \neq author \end{cases}$$

Let $\text{Xml}''_{types} = \{doc^{(0)}, el^{(0)}, att^{(0)}, tx^{(1)}\}$. Then we have $\Sigma' = \text{Xml}''_{types} \cup \{author^{(0)}\}$ and $\Delta' = \text{Utf8}$. The relabeling lifted to data trees rel_α maps the data tree $t_{2.1}$ in Figure 2.1 to the unary data tree in Figure 2.2.

Another useful relabeling that preserves all information while abstracting general data trees to data trees where nodes have at most one data value, is abstraction rel_{concat} that concatenates the data values, if more than one present, using a special separation symbol $\diamond$:

$$concat(\sigma^{(k)}(w_1, \ldots, w_k)) = \begin{cases} \sigma^{(1)}(w_1 \diamond \ldots \diamond w_k) & \text{if } k \geq 1 \\ \sigma^{(0)} & \text{else} \end{cases}$$

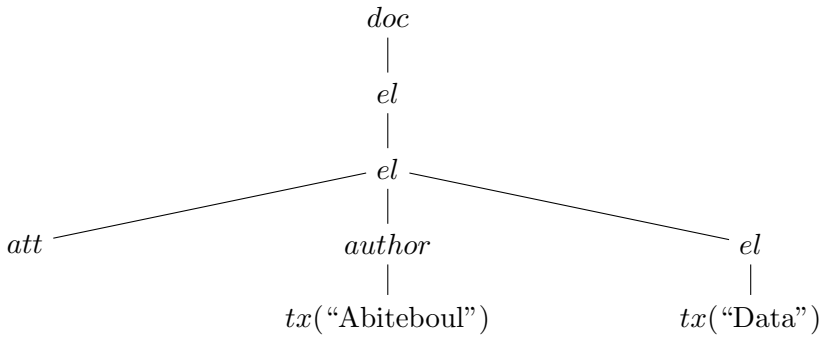


Figure 2.2: The abstracted data tree $rel_\alpha(t_{2.1}) \in \mathcal{T}_{\text{XML}''_{types} \cup \{author^{(0)}\}, \text{UTF8}}$ where only author tags and text values are preserved.

2.1.4 Json Trees

The Json format [Crockford 2001] contains objects and arrays, which in turn are made up of values. Values are either strings, numbers, objects, arrays, booleans, or null. Objects are unordered sets of key/value pairs where no two pairs contain the same key, and arrays are ordered lists of values. Keys are given by strings. We give a sample Json document that describes a person in Figure 2.3. We display objects in curly brackets, arrays in square brackets, strings in quotes, and for numbers and booleans we just state the respective value.

Data trees that underly Json have signature

$$\Sigma = \{obj^0, arr^0, str^1, numb^1, bool^1, null^0\}.$$

A Json data tree is given in Figure 2.4. However the logical structure of Json data trees is slightly different than for general data trees in that relations ns, ns^+, ps, ps^+, fo, pc, and fut are not defined on children of obj-labeled nodes, since Json objects are unordered sets. Furthermore since keys must be unique in Json, we also must impose this constraint on data values of children of obj-labeled nodes.

We next discuss how Json data trees and data trees as in Xml can be identified with each other. Starting from a data tree as in Xml, we have two possibilities to encode it into a Json data tree: We can either pack each child of a node into a new “array”-node with label arr while preserving the order of the children. Or we first annotate each node by its position, i.e. the i-th child of a node is annotated by i, and then we pack the children of a node into a new “object”-node instead of an “array”-node, with a str-labeled child with the annotation of nodes as data value. Conversely, a Json data tree can be encoded into an Xml data tree by ordering children of obj-labeled nodes according to their data value. This shows that data trees in Xml or respectively in Json are very similar.

```
{
  "name": "PPeter",
  "addr": { "street": "Hell 2",
            "city": "Heaven" },
  "age": 31,
  "phone numbers": [
    { "type": "home",
      "number": "1234567" },
    { "type": "work",
      "number": "7654321" }
  ],
  "isMale": true,
  "email": null
}
```

Figure 2.3: A JSON document.

2.2 Nested Words

2.2.1 Definition

A *nested word* [Alur 2009] is a word over three disjoint alphabets O, C, and Δ, with opening tags $\langle o \rangle \in O$, closing tags $\langle /c \rangle \in C$, and internal letters in Δ. *Complete nested words* are nested words that are more restricted in that they are well-balanced, corresponding opening and closing tags must have the same label, and initial opening tags cannot be closed before the end. The positions of nested words are called *events*, of which there are three kinds: opening, closing, and internal. Whenever O and C have the same set of labels Σ within opening and closing parenthesis, we permit ourselves to define only Σ, instead of $O = \{\langle a \rangle \mid a \in \Sigma\}$ and $C = \{\langle /a \rangle \mid a \in \Sigma\}$.

The following nested word is a word over signatures $\Sigma = \{l, b, a, t\}$ and $\Delta = \{A, b, i, t, e, o, u, l, D, a\}$:

$$\langle l \rangle \langle b \rangle \langle a \rangle \text{Abiteboul} \langle /a \rangle \langle t \rangle \text{Data} \langle /t \rangle \langle /b \rangle \langle /l \rangle$$

Sometimes information from the past may want to be stored within the label of tags of a nested word. For example, we could annotate each opening and closing event by its number as in

$$\langle (l,1) \rangle \langle (b,2) \rangle \langle (a,3) \rangle \text{Abiteboul} \langle /(a,4) \rangle \langle (t,5) \rangle \text{Data} \langle /(t,6) \rangle \langle /(b,7) \rangle \langle /(l,8) \rangle$$

2.2.2 Linearization of Data Trees

We show how to linearize unary data trees into nested words. While data trees in general have arity $k \geq 1$, we can always abstract them to signatures whose maximal arity is 1 via abstraction rel_{concat} of before.

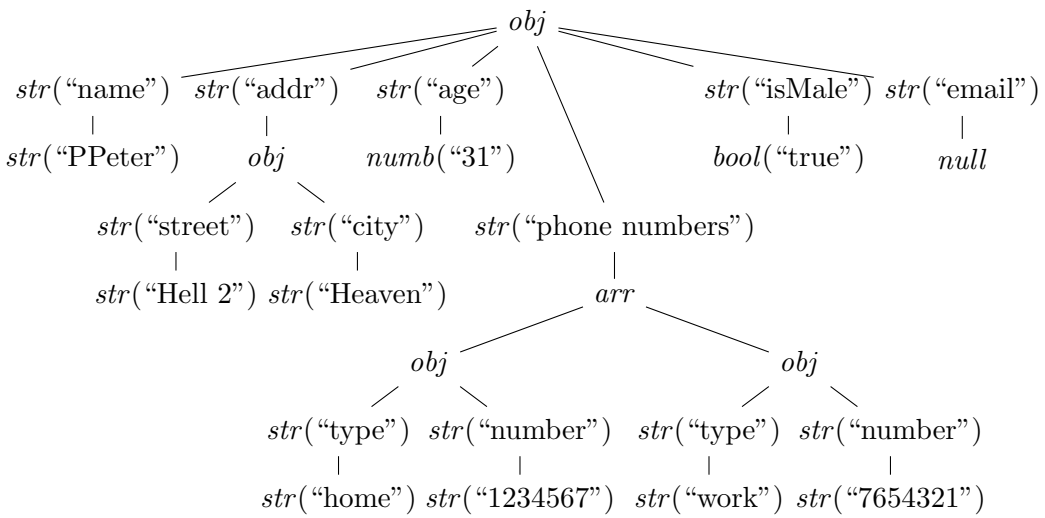


Figure 2.4: A Json data tree representing the Json document of Figure 2.3.

In order to linearize unary data trees over Σ and Δ into nested words, we need to fix the following three functions:

$$op : \mathcal{A} \to O, \quad int : \mathcal{A} \to \Delta^*, \quad \text{and} \quad cl : \mathcal{A} \to C$$

If not stated otherwise, functions *op* and *cl* will map the finite part σ of node labels to corresponding opening and closing tags, while function *int* will map to the data value w or ε if absent:

$$op(a) = \langle \sigma \rangle, \quad int(a) = \begin{cases} w & \text{if } a = \sigma^{(1)}(w) \\ \varepsilon & \text{if } a = \sigma^{(0)} \end{cases}, \quad \text{and} \quad cl(a) = \langle / \sigma \rangle$$

Once we have fixed functions *op*, *int*, and *cl* we can linearize any data tree $t = a(t_1, \ldots, t_n)$ into the following nested word $lin(t)$:

$$lin(a(t_1, \ldots, t_n)) = op(a) \cdot int(a) \cdot lin(t_1) \cdot \ldots \cdot lin(t_n) \cdot cl(a)$$

2.3 Xml Data Trees

The logical structure of an Xml document is known as the data model, which provides a tree representation of the Xml document.

2.3.1 Xml Data Model

When referring to the Xml Data model, we mean the informally described tree representation of the data model specification[1] for XPath 3.0, XQuery 3.0, and Xslt 3.0. We formalize it in the following by means of Xml data trees.

[1] http://www.w3.org/TR/xpath-datamodel-30/

The data model contains six[2] types of nodes, where in parenthesis we give their abbreviation used throughout this thesis: document (doc), element (el), attribute (att), text (tx), comment (com), processing-instruction (pi).

We model the tree-structure of the XML data model via data trees, while moving node properties as data to node labels, and moving node type information to

$$\Sigma = \text{XML}_{types} = \{doc^{(3)}, el^{(6)}, att^{(5)}, tx^{(1)}, com^{(1)}, pi^{(3)}\}.$$

In Figure 2.5 we give a grammar of data trees obtained by the XML data model together with their node properties. In gray we depict node properties that are not directly relevant when posing XPATH queries by the grammar of XPATH 3.0 [Robie 2014b] but they become relevant when looking at respective XPATH 3.0 functions[3] (for properties base-uri, document-uri, is-id, and is-idref) or for DTD processing tasks (unparsed-entities). Node property `node-name` contains the name of an element or attribute node, that consists of a namespace prefix and a local name. Property `target` holds the name of a processing-instruction, values for attribute nodes are stored in `string-value`, and property `content` contains the data of text, comment, and processing-instruction nodes. Finally, properties `nilled` and `schema-type` are needed for schema validation. `nilled` indicates that an element node should be accepted as valid when it has no content. Property `schema-type` is an XML schema type, which is string "xs:untyped" for elements and "xs:untypedAtomic" for attributes in the absence of schema.

XML data trees, i.e. trees obtained from XML documents according to the XML data model, thereby satisfy the following restrictions: The root is the only node whose type is document. All nodes of type attribute, text, comment, and processing-instruction are leafs. Only element nodes have attribute nodes as children, which must precede all its other children. These restrictions are most relevant for early query answering for XPATH. For instance, for the query `title[@lang='eng']`, any element title-labeled node can be rejected, once the first non-attribute child was read, and there was no `lang` attribute with string-value `eng` before.

An example of an XML document is given in Figure 2.6 and its corresponding XML data tree is the one in Figure 2.1. It stores the XML type, but only a subset of the data values from the XML datamodel: no data values for document nodes, node-name for element and attribute nodes, the attribute value for attribute nodes, and the data content for text nodes.

2.3.2 Xml Data Model Specifics

Document order. The so-called *document order* is a total order among all accessible nodes: The root node is the first node. Children and descendants of nodes

[2]Notice that we do not treat namespace nodes which are contained in the data model. This is because the namespace axis, which selects namespace nodes, is deprecated as of XPATH 2.0, and is thus not supported by us, which in turn does not require to treat namespace nodes. It is also not supported in XQUERY.

[3]https://www.w3.org/TR/xpath-functions-30/

Grammar:

$doc \rightarrow (el + pi + com + tx)^*$
$el \rightarrow att^*(el + pi + com + tx)^*$
$att \rightarrow none$
$tx \rightarrow none$
$com \rightarrow none$
$pi \rightarrow none$

Node labels:

doc(base-uri, unparsed-entities, document-uri)
el(base-uri, node-name, schema-type, nilled, is-id, is-idrefs)
att(string-value, node-name, schema-type, is-id, is-idrefs)
tx(content)
com(content)
pi(base-uri, target, content)

Figure 2.5: Grammar and node labels by Xml data model abstracted data trees.

```
<library>
    <book id="bk231">
        <author>Abiteboul</author>
        <title>Data</title>
    </book>
</library>
```

Figure 2.6: An Xml document.

occur after the respective node, while they occur before following siblings to the right. Attribute nodes if present follow immediately their element parent nodes. The relative order of attribute nodes is implementation-dependent, in that an implementation may chose the order, but it does not change during the processing of a query or transformation.

String-values. The data model defined the string-value of attribute nodes to be the value of the attribute, while the string-value for document and element nodes is the concatenation of its text nodes descendants in document order. Notice that this value can be derived for document and element nodes, for which reason we do not store it as data value in Xml data trees.

Children of element nodes. The Xml data model describes that children of element nodes may exclusively be element, processing-instruction, comments, and text nodes, while attributes, and document nodes can never appear as a child of an element node. Even though that the attributes of an element are not considered to be children, the element is considered to be the parent of these attribute nodes

$$
\begin{array}{l}
\langle doc\rangle \\
\quad \langle el\rangle library \\
\quad\quad \langle el\rangle book \\
\quad\quad\quad \langle att\rangle id \diamond bk231 \langle /att\rangle \\
\quad\quad\quad \langle el\rangle author \\
\quad\quad\quad\quad \langle tx\rangle Abiteboul \langle /tx\rangle \\
\quad\quad\quad \langle /el\rangle \\
\quad\quad\quad \langle el\rangle title \\
\quad\quad\quad\quad \langle tx\rangle Data \langle /tx\rangle \\
\quad\quad\quad \langle /el\rangle \\
\quad\quad \langle /el\rangle \\
\quad \langle /el\rangle \\
\langle /doc\rangle
\end{array}
$$

Figure 2.7: The nested word obtained from the XML data tree in Figure 2.1.

however.

Data trees by the XML data model have a different definition of the child relation, where attribute nodes are children of element nodes that precede any other children. Therefore when answering XPATH queries on such data trees, one has to be careful, that the child axes are translated correctly, as to come in Section 7.3.

2.3.3 Mapping to Nested Words and Back to Xml Documents

From an XML data tree we can produce a nested word by the linearization of data trees as in Section 2.2.2. This requires abstraction to maximal unary signature XML data trees obtained by concatenation of data values using a special separation symbol. Such nested words can be interpreted as XSLT programs. The nested word obtained from the XML data tree in Figure 2.1 is given in Figure 2.7. In turn it can be interpreted as a representation of the XSLT program in Figure 2.8. Indeed, this XSLT program evaluates to the original XML document in Figure 2.6.

We summarize the correlations XML documents, data trees, and nested words via the bijections depicted in Figure 2.9. It allows to identify the nested word for some XML document D by

$$nw(D) = lin(rel_{concat}(\text{data-model}(D))).$$

Furthermore $nw(D)$ allows to retrieve the XML documents D again by interpreting the nested word as an XSLT program that creates D:

Lemma 1. *For all* XML *documents* D *it holds that:*

$$D = evaluate(asXSLT(nw(D)))$$

The notion of nested words thus formally captures the essence of an XML document, while ignoring the syntactic details of the XML standard. While all our

```
<xsl:document>
    <xsl:element name="library">
        <xsl:element name="book">
            <xsl:attribute name="id">bk231</xsl:attribute>
            <xsl:element name="author">
                <xsl:text>Abiteboul</xsl:text>
            </xsl:element>
            <xsl:element name="title">
                <xsl:text>Data</xsl:text>
            </xsl:element>
        </xsl:element>
    </xsl:element>
</xsl:document>
```

Figure 2.8: The XSLT program represented by the nested word in Figure 2.7: it evaluates to the original XML document in Figure 2.6.

practical tools will process concrete XML documents, our algorithms can assume that these come as nested words.

2.4 Nested Word Automata

Our finite state technology for processing words will be nested word automata [Alur 2009].

2.4.1 Definition

A nested word automaton (NWA) is a pushdown automaton that runs on nested words and suffixes. The usage of the pushdown of an NWA is restricted: a single symbol is pushed at opening tags, a single symbol is popped at closing tags, and the pushdown remains unchanged when processing internal letters. More formally:

Definition 1. *A nested word automaton is a tuple* $A = (O, C, \Delta, Q, Q_I, Q_F, \Gamma, R)$ *where* O*,* C*, and* Δ *are the finite alphabets of nested words,* Q *a finite set of states with subsets* $Q_I, Q_F \subseteq Q$ *of initial and final states,* Γ *a finite set of stack symbols, and* R *is a set of transition rules of the following three types, where* $q, q' \in Q$*,* $\langle o \rangle \in O$*,* $\langle /c \rangle \in C$*, and* $d \in \Delta$*:*

(open) $q \xrightarrow{\langle o \rangle : \gamma} q'$ *can be applied in state* q*, when reading the opening parenthesis* o*. In this case,* γ *is pushed onto the stack and the state is changed to* q'*.*

(close) $q \xrightarrow{\langle /c \rangle : \gamma} q'$ *can be applied in state* q *when reading the closing parenthesis* c *with* γ *on top of the stack. Then,* γ *is popped and the state is changed to* q'*.*

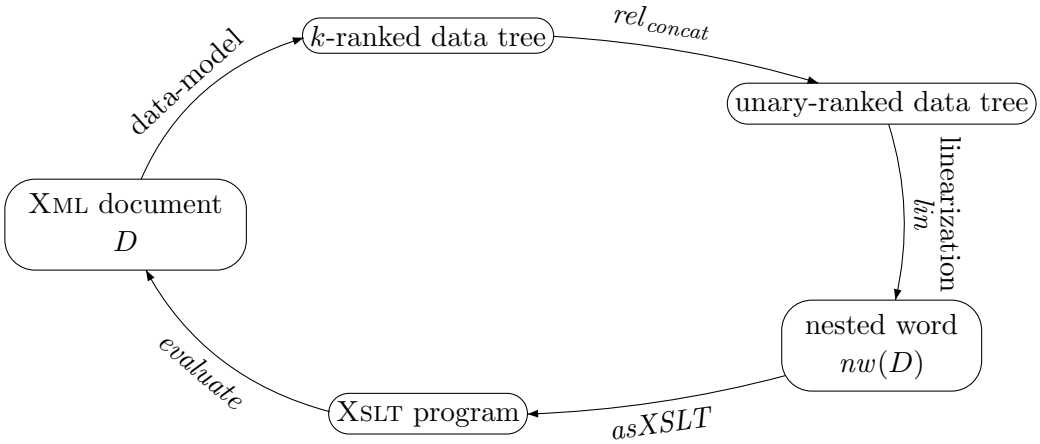


Figure 2.9: XML documents D as $nw(D) = lin(rel_{concat}(\text{data-model}(D)))$, and back $D{=}evaluate(asXSLT(lin(rel_{concat}(\text{data-model}(D)))))$.

(internal) $q \xrightarrow{d} q'$ *can be applied in state* q *when reading the internal letter* d*. One then moves to state* q'*.*

An NWA is called *deterministic* or a dNWA if it is deterministic as a pushdown automaton. In contrast to more general pushdown automata, NWAs can always be determinized [Alur 2009], essentially, since they have the same expressiveness as bottom-up tree automata.

2.4.2 Runs of Nested Words

A configuration of an NWA is a state-stack pair in $Q \times \Gamma^*$. Let $S \in \Gamma^n$ be a stack of depth $n \geq 0$ and s be a suffix of a complete nested word over O, C, and Δ, which has excess $-n$. An S-run on s must start in a configuration with stack S and some initial state, and then rewrite this configuration on all events of the suffix according to some of the rules in R. An S-run on a suffix is called *successful* if it continues until the end while reaching some final state. Note that the stack will always be empty at the end of suffixes, given that the depth of S had the same absolute value as the excess of the suffix and given the restriction on pushdowns of NWAs (often called "visible"). The *language* $\mathcal{L}_S(A)$ of an NWA A is the set of all suffixes over O, C, and and Δ that permit a successful S-run by A.

An example of a run of the NWA in Figure 2.10 on a suffix of a complete nested word is depicted in Figure 2.11 with start stack $S = \alpha$. The automaton tests whether the suffix of the nested word as a tree satisfies the XPATH filter `[descendant::a[descendant::b]]`. When opening an `a`-node in its initial state q_0 the NWA guesses that it matches the `a`-descendant of the XPATH query, it pushes α onto the stack and moves to state q_1. From there it waits until the guess of the `b`-descendant in the query, where it moves into final state q_3, where it stays until the end.

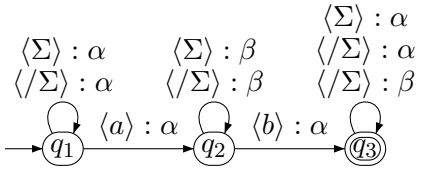


Figure 2.10: An NWA where $\Sigma = \{a, b\}$ and $\Delta = \emptyset$.

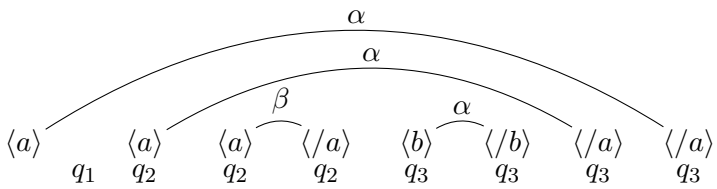


Figure 2.11: A run of the NWA of Figure 2.10 on a suffix of a nested word.

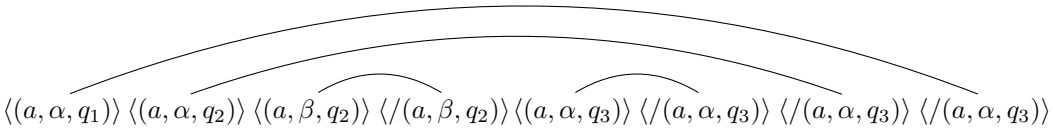


Figure 2.12: The run of Figure 2.11 as a nested word.

Furthermore the runs of nested words can be identified with nested words themselves. In Figure 2.12 we show the run of Figure 2.11 as a nested word. However, corresponding opening and closing tags are labeled differently: For example opening tag $\langle(a, \alpha, q_2)\rangle$ does not have the same label as its corresponding closing tag $\langle/(a, \alpha, q_3)\rangle$. Such nested words are still linearizations of trees, but with functions $op((a, \alpha, q, q')) = \langle(a, \alpha, q)\rangle$ and $cl((a, \alpha, q, q')) = \langle/(a, \alpha, q')\rangle$. Since runs of NWAs are nested words, as a consequence, NWAs can be composed.

2.4.3 Runs on Marked Trees

A *marked data tree* is a pair (t, π) consisting of a data tree t and a node $\pi \in \mathit{Nodes}(t)$, called the *mark*. Marked unary data tree (t, π) can be linearized into the nested word, which is the suffix of the complete nested word by the linearization of t starting at the opening event of π. More formally, we define for any data tree $t = a(t_1, \ldots, t_n)$ that contains node π a tree suffix $\mathit{suff}(t, \pi)$ as follows:

$$\mathit{suff}(a(t_1, \ldots, t_n), \varepsilon) = \mathit{lin}(a(t_1, \ldots, t_n))$$
$$\mathit{suff}(a(t_1, \ldots, t_n), i\pi) = \mathit{suff}(t_i, \pi) \cdot \mathit{lin}(t_{i+1}) \cdot \ldots \cdot \mathit{lin}(t_n) \cdot \mathit{cl}(a)$$

The marked node of a tree suffix is the marked node of the underlying marked data tree. Tree suffixes are complete nested words if and only if the marked node is the root. More generally, tree suffixes lack the opening events for all ancestors of the marked node.

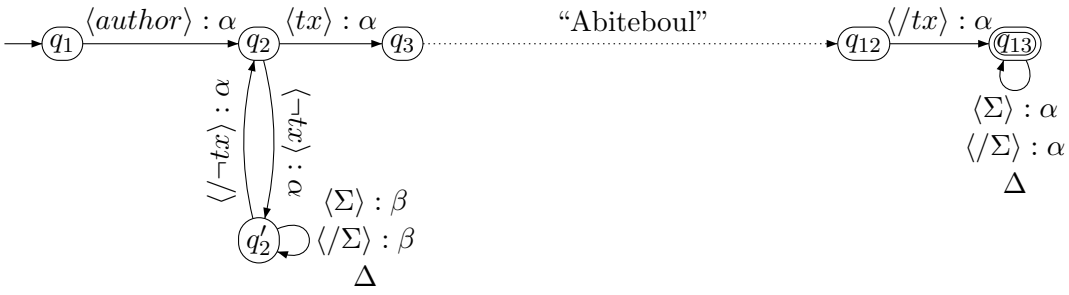


Figure 2.13: NWA for XPATH filter `[author[child::text()[1] = "Abiteboul"]]` with shortcuts $\Sigma = \text{XML}''_{types} \cup \{author^{(0)}\}$, $\neg tx = \Sigma \setminus \{tx\}$, and $\Delta = \text{UTF8}$.

An example for a marked data tree is given in Figure 2.14, where the `author`-node is chosen as the marked node, and thereby underlined. The NWA in Figure 2.13 answers the XPATH filter `[author[child::text()[1] = "Abiteboul"]]` and thereby accepts all tree suffixes $\mathit{suff}(t, \pi)$ for a marked data tree (t, π) of trees t obtained from XML documents and a marked `author`-node π, whose first text node's data content is equal to "Abiteboul". A successful run over the marked data tree is given in Figure 2.14 with start stack $S = \alpha\alpha\alpha$.

2.4.4 Determinization of NWAs

In the following we give a direct determinization procedure for nested word automata over alphabets Σ and Δ. These are more restricted nested word automata in that the alphabets O for opening and C for closing parenthesis coincide to $\Sigma = O = C$. It is possible to determinize NWAs in the general case. However the determinization becomes more complicated and moreover the more general case is not needed for our algorithms.

An NWA $A = (\Sigma, \Delta, Q, Q_I, Q_F, \Gamma, R)$ is *deterministic* if Q_I contains at most one state, and for any $q \in Q$, $a \in \Sigma$, and $a' \in \Delta$ there is at most one pair (γ, q') with $q \xrightarrow{\langle a\rangle:\gamma} q' \in R$, for any $\gamma \in \Gamma$ there exists at most q' such that $q \xrightarrow{\langle/a\rangle:\gamma} q' \in R$, and there exists at most one q' such that $q \xrightarrow{a'} q' \in R$.

Clearly, any dNWA permits at most one run per tree. The difficulty is to deal with competing opening rules $q \xrightarrow{\langle a\rangle:\gamma_1} q_1$ and $q \xrightarrow{\langle a\rangle:\gamma_1} q_2$. Essentially, the choice is delayed until closing time. The states and stack symbols of the determinization of A are sets of such pairs of states $(q, q') \in Q^2$, with the intuitive meaning that the sequence of children of the current node can be evaluated to q' when starting in q.

The three inference rules below define its transitions.

$$\frac{\bar{Q}' = \{(q_3, q_3) \mid (q_1, q_2) \in \bar{Q}, q_2 \xrightarrow{\langle a\rangle:\gamma} q_3 \in R\}}{\bar{Q} \xrightarrow{\langle a\rangle:\bar{Q}} \bar{Q}'}$$

At opening time the current set $\bar{Q}$ is pushed onto the stack and the evaluation of the sequence of the children of the current node is started in all pairs (q_3, q_3), for

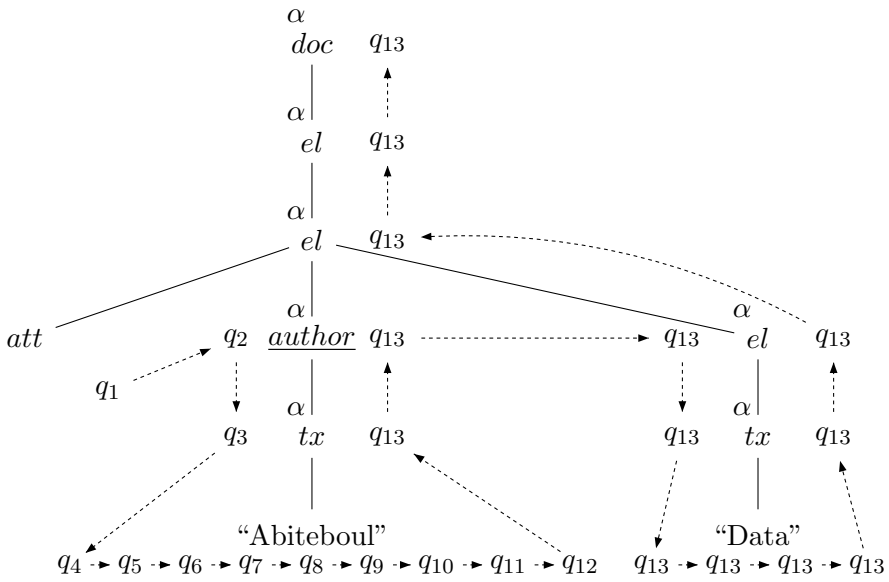


Figure 2.14: A successful run of the NWA of Figure 2.13 on the tree suffix of abstracted data tree from Figure 2.2 where the `author`-node is chosen as the mark. Here the start stack is equal $S = \alpha\alpha\alpha$ containing stack symbol α for every ancestors of the marked node.

which q_3 can be reached by some transition of A.

$$\frac{\bar{Q}'' = \left\{ (q_1, q_5) \mid \begin{array}{l} (q_3, q_4) \in \bar{Q},\ (q_1, q_2) \in \bar{Q}', \\ q_2 \xrightarrow{\langle a \rangle : \gamma} q_3, q_4 \xrightarrow{\langle / a \rangle : \gamma} q_5 \in R \end{array} \right\}}{\bar{Q} \xrightarrow{\langle / a \rangle : \bar{Q}'} \bar{Q}''}$$

At closing time, if one of the left siblings of the a-node could be evaluated to (q_1, q_2) and the sequence of children of the a-node to (q_3, q_4) then one can evaluate the sequence of left siblings including the a-subtree to (q_1, q_5), if q_2 can be opened to q_3 with the same stack symbol γ that allows to close q_4 to q_5.

$$\frac{\bar{Q}' = \{(q_1, q_3) \mid (q_1, q_2) \in \bar{Q}, q_2 \xrightarrow{a} q_3 \in R\}}{\bar{Q} \xrightarrow{a} \bar{Q}'}$$

The internal rule allows to only update the current state in the second component of all pairs of $\bar{Q}$.

The only initial state of the determinization of A is $\{(q, q) \mid q \in Q_I\}$. A state $\bar{Q}$ of the determinization of A is final if it contains some pair (q, q') with $q \in Q_I$ and $q' \in Q_F$.

In the worst case, the determinized automaton has $2^{|Q|^2}$ states. In experiments, we also observed huge size explosions in the average case. For example, for the XPath query `//a[following-sibling::b[.//c][./d]]/e` we obtain an NWA with 38 states and 7719 transitions. We were not able to construct the corresponding

dNWA even if restricted to accessible states only. The main limiting factor was memory. For the mentioned query we stopped the construction shortly, after having reached 5000 states with more than 20 million transitions, and swapping to the disk. Therefore, we will mostly rely on on-the-fly determinization.

2.5 Recursive Functions

The semantics of definitions of recursive functions is usually based on fixed points on complete partial orders. We first recall the Kleene fixpoint theorem that extracts the conditions under which such fixpoints exist, and then introduce lambda notion for recursive functions based on this fixed point theorem.

2.5.1 Partial Orders and Fixed Points.

A *partial order* on a set D is a binary relation $\sqsubset$ on D which is irreflexive, anti-symmetric, and transitive. For any partial order let $\sqsubseteq$ be the reflexive closure of $\sqsubset$.

Definition 2. *We call a partial order complete or equivalently a* CPO *if for any ascending chain $v_1 \sqsubseteq v_2 \sqsubseteq \ldots$ of D has least upper bound $\sqcup\{v_i \mid i \in \mathbb{N}\} \in D$.*

We call a function $f : D \to D$ *monotone* if for all $v_1, v_2 \in D$:

$$v_1 \sqsubseteq v_2 \Rightarrow f(v_1) \sqsubseteq f(v_2)$$

We call f *Scott-continuous* if every ascending chain $v_1 \sqsubseteq v_2 \sqsubseteq v_3 \ldots$ of elements in D has an upper bound such that:

$$f(\sqcup\{v_n \mid n \in \mathbb{N}\}) = \sqcup f(\{v_n \mid n \in \mathbb{N}\}).$$

We recall that any Scott-continuous function is monotone. A fixed point of f is an element v of D such that $f(d) = d$.

Theorem 1 (Kleene fixed-point theorem)**.**
Let $(D, \sqsubset)$ be a complete partially ordered set with least element $\bot$ and $f : D \to D$ a Scott-continuous function. Then f has a least fixed point $lfp_f = \sqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$.

Proof. Since $\bot$ is the least element in D, we have that $\bot \sqsubseteq f(\bot)$. Since f is monotone, we have that $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ for all $n \geq 0$. The chain has a supremum since $(D, \sqsubset)$ is complete, so $lfp_f = \sqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is well defined. It follows that lfp_f is a fixed point of f since f is Scott continuous (and thus monotone):

$$f(lfp_f) = f(\sqcup\{f^n(\bot) \mid n \in \mathbb{N}\}) = \sqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\} = lfp_f$$

Again from monotonicity it follows that lfp_f is smaller than any other fixed point of f. □

2.5.2 Lambda and Letrec Notation

We next introduce Λ notation for total functions, which is situated on the meta-level of mathematics, rather than on the formal language level such as λ-terms.

Definition 3. *Let D, D' be sets and $f(v) \in D'$ for any $v \in D$. Then we write*

$$\Lambda\ v \in D.\ f(v)$$

for the total function $\{(v, f(v)) \mid v \in D\}$.

We next introduce **Letrec** notation for recursive functions, by which we explain how recursive functions can be defined by least fixed points and can be computed functional programs.

Here we will have to deal with partial functions $g \subseteq D \times D'$, which form a CPO with the inclusion ordering. Its least element is the empty partial function $\emptyset$.

Definition 4. *Let $f = (f_1, \ldots, f_n)$ be a tuple of total function such that f_i returns for any input tuple $v = (v_1, \ldots, v_n)$ of partial functions $v_i \subseteq D_i \times D'_i$ an output tuple $f(v) = (f_1(v), \ldots, f_n(v))$ of partial functions $f_i(v) \subseteq D_i \times D'_i$. If f is Scott-continuous on the* CPO*:*

$$(D_1 \times D'_1) \times \ldots \times (D_n \times D'_n)$$

then we write:

$$\textbf{Letrec}\ v_1 = f_1(v) \ldots v_n = f_n(v)\ \textbf{in}\ v$$

for the least fixed point of the operator f, which exists by Kleene's fixed point theorem (Theorem 1) and has the following form, since the tuple of empty partial relations is the least element of the above CPO*:*

$$\sqcup\{f^n(\emptyset, \ldots, \emptyset) \mid n \in \mathbb{N}\}.$$

Furthermore, if there is a function g which can be applied to all elements of the above CPO *then we extend the* **Letrec** *notation as follows:*

$$\begin{aligned} &\textbf{Letrec}\ v_1 = f_1(v) \ldots v_n = f_n(v)\ \textbf{in}\ g(v) \\ &\quad =_{df}\ g(\textbf{Letrec}\ v_1 = f_1(v) \ldots v_n = f_n(v)\ \textbf{in}\ v) \end{aligned}$$

Proposition 1. *Let $f = (f_1, \ldots, f_n)$ be a tuple of functions as above and $(F_1, \ldots, F_n)$ a tuple of functional programs such that F_i computes f_i. Then for any $1 \leq i \leq n$, there exists an functional program that computes the function in the i-th component of $v = (v_1, \ldots, v_n)$ where:*

$$\textbf{Letrec}\ v_1 = f_1(v) \ldots v_n = f_n(v)\ \textbf{in}\ v$$

Proof. Let *lfp* be the least fixed point. The idea is to unfold the definitions of the functions f_i as far as needed. This can be done by the following recursively defined functional program:

$$
\begin{array}{l}
\mathit{lfp} = \\
\quad \textbf{fun}(j) \\
\quad\quad \textbf{fun}(v) \\
\quad\quad\quad \textbf{case } j \\
\quad\quad\quad\quad \textbf{of } 1 \textbf{ then } F_1(\mathit{lfp}\ 1, \ldots, \mathit{lfp}\ n)(v) \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad \textbf{of } n \textbf{ then } F_n(\mathit{lfp}\ 1, \ldots, \mathit{lfp}\ n)(v) \\
\textbf{in } \mathit{lfp}(i)
\end{array}
$$

□

We next illustrate the proof of the proposition for the functional program computing the following two mutual recursive functions:

$$
\begin{array}{l}
\textbf{Letrec} \\
\quad v_1 = \Lambda\ v \in \mathbb{N}.\ v_2\ v - 1 \\
\quad v_2 = \Lambda\ u \in \mathbb{N}.\ \textbf{if } u = 0 \textbf{ then } 1 \textbf{ else } v_1\ u \\
\textbf{in } v_2\ 3
\end{array}
$$

Here we use the functions f_1 and f_2 such that for all v_1, v_2:

$$
\begin{array}{l}
f_1(v_1, v_2) = \Lambda\ v \in \mathbb{N}.\ v_2\ v - 1 \\
f_2(v_1, v_2) = \Lambda\ u \in \mathbb{N}.\ \textbf{if } u = 0 \textbf{ then } 1 \textbf{ else } v_1\ u
\end{array}
$$

The proof of Proposition 1 yields the following functional program for computing the components of **Letrec** $v_1 = f_1(v_1, v_2)\ v_2 = f_2(v_1, v_2)$ **in** (v_1, v_2):

$$
\begin{array}{l}
\mathit{lfp} = \textbf{fun } j. \\
\quad\quad \textbf{fun } v. \\
\quad\quad\quad \textbf{case } j \\
\quad\quad\quad\quad \textbf{of } 1 \textbf{ then } \mathit{lfp}\ 2\ v - 1 \\
\quad\quad\quad\quad \textbf{of } 2 \textbf{ then if } v = 0 \textbf{ then } 1 \textbf{ else } \mathit{lfp}\ 1\ v
\end{array}
$$

When applied to the second function with argument 2, we obtain

$$
\begin{array}{rcl}
\mathit{lfp}\ 2\ 3 & = & \textbf{if } 3 = 0 \textbf{ then } 1 \textbf{ else } \mathit{lfp}\ 1\ 3 \\
& = & \mathit{lfp}\ 1\ 3 \\
& = & \mathit{lfp}\ 2\ 3 - 1 \\
& = & \mathit{lfp}\ 2\ 2 \\
& = & \textbf{if } 2 = 0 \textbf{ then } 1 \textbf{ else } \mathit{lfp}\ 1\ 2 \\
& = & \mathit{lfp}\ 1\ 2 \\
& = & \mathit{lfp}\ 2\ 2 - 1 \\
& = & \mathit{lfp}\ 2\ 1 \\
& = & \textbf{if } 1 = 0 \textbf{ then } 1 \textbf{ else } \mathit{lfp}\ 1\ 1
\end{array}
$$

$$
\begin{array}{ll}
= & \mathit{lfp}\ 1\ 1 \\
= & \mathit{lfp}\ 2\ 1 - 1 \\
= & \mathit{lfp}\ 2\ 0 \\
= & \textbf{if}\ 0 = 0\ \textbf{then}\ 1\ \textbf{else}\ \mathit{lfp}\ 1\ 0 \\
= & 1
\end{array}
$$

Part I

A Formal Semantics of XPath 3.0

Chapter 3

Navigational XPath 3.0

Contents

In this chapter, we distinguish the navigational XPath 3.0 fragment that we call Navigational XPath for short.

3.1 Overview

The XPath queries that we present in the following apply to Xml data trees that satisfy the Xml data model that we discussed in Section 2.3.1. Navigational queries select sets of nodes of Xml data trees that are stored in a **sequence**. The default semantics claims that nodes are to be sorted into document order, while an arbitrary implementation dependent order can be chosen within the *unordered* semantics of XQuery.

While we explain in detail in Chapter 4 the types and values of XPath 3.0, for this chapter it suffices to understand **sequences** as a flat ordered list that contains **items** which can either be a node, an atomic value, or a function. However, the result of path expressions by navigational queries are always sequences of nodes, while filters evaluate to a boolean or a sequence of nodes which is to be tested for non-emptiness.

Navigational XPath is obtained by restricting the grammar of XPath 3.0 [Robie 2014b]. Whenever a rule (X) of the grammar is restricted, the remaining

rules is given the name (X'). The restricted grammar that we present in the next section allows for navigation in XML data trees by forward and backward axis, with relative and absolute expressions, filters, and compositions of paths. For example, a navigational XPATH query is

```
descendant::a[child::b]/parent::c
```

that selects the `c`-labeled parent of `a`-descendants that have a `b`-child. This query contains filters and path compositions of single forward and backward paths. Furthermore, navigational queries allow for more complex tests on node labels and types, such as in query

$$child :: p : a/child :: text()$$

that selects all text children of type tx, below an `a`-labeled child of the root with namespace prefix `p`. However, any data value comparisons, arithmetics, function application are excluded by navigational XPATH.

3.2 Grammar

We give the grammar of navigational XPATH that we obtain from the grammar of XPATH 3.0 [Robie 2014b] via some minor modifications. We leave out certain rules in order to disallow certain expressions. For the same reason we change slightly some grammar rules, for example, rule (1) becomes rule (1'). Finally we add three rules for negation (rule (Neg)) and boolean values *true* (rule (True)) and *false* (rule (False)). Note that these exist in XPATH 3.0 as built-in functions `fn:not`, `fn:true`, and `fn:false`, but here we add them to the grammar, since no other functions are needed.

3.2.1 Ebnf

The grammar of navigational XPATH is given in Extended Backus-Naur Form (EBNF). There all nonterminals have a name that starts with a capital letter, while terminal symbols are the strings within double quotes. We refer to the specification[1] for more details about the EBNF and describe only a few patterns `P` in the following. A pattern `P` appears in the right-hand side of a grammar rule, that is either a terminal, a nonterminal, or which consist of the more complex patterns, where `P` and `P'` are sub-patterns:

- `(P)` matches `P`,
- `P?` matches `P` or nothing,
- `P P'` matches `P` followed by `P'` ("space" pattern),
- `P | P'` matches `P` or `P'` ("or" pattern),

[1] https://www.w3.org/TR/xpath-30/#EBNFNotation

- `P - P'` matches all strings that match `P`, but not `P'`,
- `P+` matches one or more occurrences of `P`,and
- `P*` matches zero or more occurrences of `P` ("star" pattern)

For example, nonterminal `RelativePathExpr` has EBNF rule

```
RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
```

which uses "space", "or", and "star" patterns. Since a "space" pattern matches the pattern before the space followed by the pattern after the space and since "star" patterns match zero ore more occurrences, any string by `RelativePathExpr` is the concatenation of a string by `StepExpr` followed by zero or more strings by `StepExpr` that are separated by characters `"/"` or `"//"`.

3.2.2 Expressions

The grammar starts with path expressions.

```
(1')   XPath             ::= OrExpr
(16)   OrExpr            ::= AndExpr ( "or" AndExpr )*
(17')  AndExpr           ::= NegationExpr ( "and" NegationExpr )*
(Neg)  NegationExpr      ::= TrueExpr | "not(" TrueExpr ")"
(True) TrueExpr          ::= FalseExpr | "true"
(False) FalseExpr        ::= PathExpr | "false"
(35)   PathExpr          ::= ("/" RelativePathExpr?)  | ("//"
                             RelativePathExpr) | RelativePathExpr
(36')  RelativePathExpr  ::= AxisStep (("/" | "//") AxisStep)*
(38)   AxisStep          ::= (ReverseStep | ForwardStep)
                             PredicateList
(39)   ForwardStep       ::= (ForwardAxis NodeTest) |
                             AbbrevForwardStep
(40)   ForwardAxis       ::= ("child" "::") | ("descendant" "::") |
                             ("attribute" "::") | ("self" "::") |
                             ("descendant-or-self" "::") |
                             ("following-sibling" "::") |
                             ("following" "::") | ("namespace" "::")
(41)   AbbrevForwardStep ::= "@"?  NodeTest
(42)   ReverseStep       ::= (ReverseAxis NodeTest) |
                             AbbrevReverseStep
(43)   ReverseAxis       ::= ("parent" "::") | ("ancestor" "::") |
                             ("preceding-sibling" "::") |
                             ("preceding" "::") |
                             ("ancestor-or-self" "::")
(44)   AbbrevReverseStep ::= ".."
(50)   PredicateList     ::= Predicate*
(51')  Predicate         ::= "[" OrExpr "]"
```

Path expression may reference node test expressions that verify node labels and kinds.

```
(45)  NodeTest              ::= KindTest | NameTest
(46)  NameTest              ::= EQName | Wildcard
(47)  Wildcard              ::= "*" | (NCName ":" "*") |
                                ("*" ":" NCName) |
                                (BracedURILiteral "*")
(71') KindTest              ::= DocumentTest | ElementTest |
                                AttributeTest | PITest | CommentTest |
                                TextTest | AnyKindTest
(72)  AnyKindTest           ::= "node" "(" ")"
(73') DocumentTest          ::= "document-node" "(" ElementTest?  ")"
(74)  TextTest              ::= "text" "(" ")"
(75)  CommentTest           ::= "comment" "(" ")"
(77)  PITest                ::= "processing-instruction" "("
                                (NCName | StringLiteral)?  ")"
(78)  AttributeTest         ::= "attribute" "(" (AttribNameOrWildcard
                                ("," TypeName)?)?  ")"
(79)  AttribNameOrWildcard  ::= AttributeName | "*"
(82)  ElementTest           ::= "element" "(" (ElementNameOrWildcard
                                ("," TypeName "?"?)?)?  ")"
(83)  ElementNameOrWildcard::= ElementName | "*"
```

Node label tests reference names.

```
(86)  AttributeName    ::= EQName
(87)  ElementName      ::= EQName
(89)  TypeName         ::= EQName
(94)  EQName           ::= QName | URIQualifiedName
(99)  URIQualifiedName::= BracedURILiteral NCName
(100) BracedURILiteral::= "Q" "{" [^{}]* "}"
(104) QName            ::= [http://www.w3.org/TR/REC-xml-names
                           /#NT-QName]
(105) NCName           ::= [http://www.w3.org/TR/REC-xml-names
                           /#NT-NCName]
```

3.2.3 Parse Tree

An example is the parse tree of XPath query `descendant::tweet` in Figure 3.1. This parse tree is obtained starting with nonterminal `XPath`. From there the only choice is nonterminal `OrExpr`. Since the query does not contain the keyword `or`, the star pattern of nonterminal `OrExpr` matched zero occurrences. So one expends into nonterminal `AndExpr`, and so on.

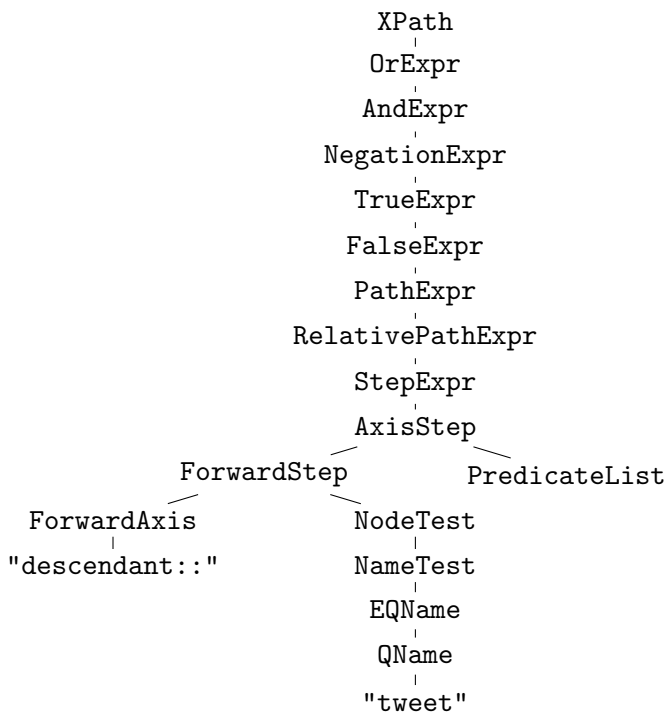


Figure 3.1: The parse tree of XPath query `descendant::tweet`.

3.3 Path Expressions

Path expressions access nodes of Xml data trees. Navigation within an Xml data tree is supported via axes by nonterminal `AxisStep` (38). They are single path steps that perform navigation via forward and backward axis to some node which is tested for its label, kind, and against a list of filters. These path steps can then be composed.

3.3.1 Forward and Backward

Single path steps use either a forward or a backward axis. Forward axes contain future nodes of the start node, i.e. those nodes that are related to the start node via future relation *fut*. In the grammar production below such an axis is `child` with proceeding colons "`::`", to which a test on the target node follows.

(40)
```
ForwardAxis::= ("child" "::") | ("descendant" "::") |
               ("attribute" "::") | ("self" "::") |
               ("descendant-or-self" "::") |
               ("following-sibling" "::") |
               ("following" "::") | ("namespace" "::")
```

Backward axes contain all other nodes, i.e. those related to the start node via relation *pc*.

(43) `ReverseAxis::= ("parent" "::") | ("ancestor" "::") | ("preceding-sibling" "::") | ("preceding" "::") | ("ancestor-or-self" "::")`

3.3.2 Label Tests

Label tests inspect the node-name property of element and attribute nodes, that is stored as data value. It represents a `QName`, which is made up of a namespace prefix and a local part (the actual label). `NameTests` by nonterminal `NameTest` (46), allow to test either the namespace prefix, the local part, or both of the nodes label.

An example is query

`descendant::a`

that selects all `a`-labeled descendants of the start node, while query

`descendant::xml:*`

selects all descendants nodes, whose namespace prefix is equal to `xml`.

3.3.3 Kind Tests

Any node can be tested for its kind, i.e. the XML type, that is also stored in node labels of XML data trees. Kinds test are the document node, element, attribute, processing-instruction, comment, and text node tests, or may be the `AnyKindTest` `node()` the matches any node kind.

(71') `KindTest::= DocumentTest | ElementTest | AttributeTest | PITest | CommentTest | TextTest | AnyKindTest`

For example, query

`child::text()`

selects all children of type *tx*. One can also test whether descendant of some node are element nodes whose label is "`a`" as in query

`descendant::element(a)`

which is equivalent to `descendant::a`.

3.3.4 Relative, Absolute, and Abbreviated Expressions

Path expressions can be relative such that they are evaluated from some node as we have seen with queries `descendant::a` and `child::text()`. Otherwise they are absolute and thereby evaluated from the root of the tree. In XPATH absolute expressions have "`/`" or "`//`" prepended to them, as in query `/descendant::a` that selects all `a`-descendants of the root node.

Most often XPATH queries use the abbreviation "`/`" for the child axis, and also when the axis name is omitted for a path step, the child axis is the default one:

`a/b` = `child::a/child::b`

An occurrence of the abbreviation "//" in XPath queries which is not at the beginning of a path expression is an abbreviation for `/descendant-or-self::node()/`, as in query

`/a//b` = `/a/descendant-or-self::node()/b`.

3.3.5 Filters

Filters a.k.a **predicates** in XPath are test expressions given in square brackets. For some node they can test whether it is related to certain other nodes. The following query, for example, retains all `book`- labeled descendant nodes for which the filter becomes true:

```
//book[not(child::title)]
```

It selects all `book`-descendants of the root, that do not have a `title`-labeled child node. Therefore, filters determine which elements of an input **sequence** are kept and which are not.

In general a filter expression `P[P']` is evaluated as follows. `P` evaluates to a **sequence** of **items**, and not necessarily to a **sequence** of nodes only. For every **item** in the **sequence** by `P` the filter `P'` is evaluated. If the filter evaluates to true, the **item** is retained. Note that filters in the original XPath grammar [Robie 2014b] allow to evaluate to an integer as well. In that case the **item** is retained if the integer is equal to the position of the **item** in the **sequence**, as with query `child::author[2]` that selects the second `author`-child.

3.3.6 Compositions

Single path steps with optional filters can be composed using XPath's path operator "/". For all `a`-children that have a `b`-child, for example, as in query

```
/child::a[child::b]/child::c
```

the expression `child::c` is evaluated, resulting in the selection of all `c`-nodes below `a`-children of the root that have a `b`-child.

In general, expressions `P` in path compositions `P/P'` must evaluate to a **sequence** on nodes. Expression `P'` is evaluated for every node in that **sequence** and result **sequences** are appended. In the original rule of the XPath 3.0 grammar of [Robie 2014b] path compositions, as by nonterminal `RelativePathExpr` (36) single path steps may also return numbers and strings, and not only nodes. The specification of the path operator "/" states that the result **sequences** by `P'` are either **sequences** of non-nodes, or they are **sequences** of nodes of which duplicates are removed, and which are returned in document order. The removal of duplicates is performed in order to obtain a *set* of nodes. Otherwise with tree $a(a(a(b)))$ one would select the common `b`-descendant several times as with query

```
/descendant::a/descendant::b
```

which selects all `b`-descendants which has some `a`-ancestor of the root.

Chapter 4

Types and Values

Contents

This chapter explains the types and values of XPath 3.0 that are based on sequences: There are atomic types, node types, and function types. Respective values are atomic values, such as integers, nodes of Xml data trees, and functions. These values are called items. Then there are sequence types, whose values are sequences, which are ordered list that are made up of items and not nested. Therefore a list of pairs ((1, "one"), (2, "two"), (3, "three")), for example, would have to be encoded into sequence (1, "one", 2, "two", 3, "three"), where every two items make up for a pair.

Type checking is static and dynamic. Statically one verifies whether there is no syntax error in the query, whether the declared functions parameters do not have the same name, or whether predefined prefixes such as `xml` are not declared elsewhere. Otherwise type checking is mostly dynamic, while there exist dynamic type-conditionals, such as `instance of`, that verify whether a sequence has an expected type or not. Variables in XPath are statically typed depending on their context, functions are annotated with input and output types, while items in general may have several types because of subtyping. For all these reasons, sequence types have a concrete syntax specified by the grammar of XPath 3.0.

4.1 Sequence Type Expressions

Sequence type expressions are derived from nonterminal `SequenceType` (67). They are expressions, but also types in the XPath 3.0 language, as we will discuss.

```
(67) SequenceType          ::= ("empty-sequence" "(" ")") |
                               (ItemType OccurrenceIndicator?)
(68) OccurrenceIndicator   ::= "?" | "*" | "+"
(69) ItemType              ::= KindTest | ("item" "(" ")") |
                               FunctionTest | AtomicOrUnionType |
                               ParenthesizedItemType
(70) AtomicOrUnionType     ::= EQName
(71) KindTest              ::= DocumentTest | ElementTest |
                               AttributeTest | SchemaElementTest |
                               SchemaAttributeTest | PITest |
                               CommentTest | TextTest |
                               NamespaceNodeTest | AnyKindTest
(72) AnyKindTest           ::= "node" "(" ")"
(73) DocumentTest          ::= "document-node" "(" (ElementTest |
                               SchemaElementTest)?  ")"
(74) TextTest              ::= "text" "(" ")"
(75) CommentTest           ::= "comment" "(" ")"
(76) NamespaceNodeTest     ::= "namespace-node" "(" ")"
(77) PITest                ::= "processing-instruction" "("
                               (NCName | StringLiteral)?  ")"
(78) AttributeTest         ::= "attribute" "(" (AttribNameOrWildcard
                               ("," TypeName)?)?  ")"
(79) AttribNameOrWildcard  ::= AttributeName | "*"
(80) SchemaAttributeTest   ::= "schema-attribute" "("
                               AttributeDeclaration ")"
(81) AttributeDeclaration  ::= AttributeName
(82) ElementTest           ::= "element" "(" (ElementNameOrWildcard
                               ("," TypeName "?"?)?)?  ")"
(83) ElementNameOrWildcard::= ElementName | "*"
(84) SchemaElementTest     ::= "schema-element" "("
                               ElementDeclaration ")"
(85) ElementDeclaration    ::= ElementName
(86) AttributeName         ::= EQName
(87) ElementName           ::= EQName
(88) SimpleTypeName        ::= TypeName
(89) TypeName              ::= EQName
(90) FunctionTest          ::= AnyFunctionTest | TypedFunctionTest
(91) AnyFunctionTest       ::= "function" "(" "*" ")"
(92) TypedFunctionTest     ::= "function" "(" (SequenceType
```

```
                                ("," SequenceType)*)?  ")" "as"
                                SequenceType
(93) ParenthesizedItemType ::= "(" ItemType ")"
(94) EQName                ::= QName | URIQualifiedName
```

4.2 Basic Types

Basic types in XPath 3.0 are **item** types. It is a type that is derived by the Ebnf rule `ItemType` (69). Their values are called **items**. Basically every **item** is either an atomic value, a node, or a function.

```
(69) ItemType ::= KindTest | ("item" "(" ")") |
                  FunctionTest | AtomicOrUnionType |
                  ParenthesizedItemType
```

The **item** type `item()` is the most general **item** type: any **item** has type `item()`. Any item has the type `ParenthesizedItemType` (93) if and only if the **item** has the type that is in parenthesis.

4.2.1 Atomic Types

The XPath 3.0 data model accommodates of up to 50 different atomic types that are derived by nonterminal `AtomicOrUnionType` (70) and that we present in Figure 4.1.

For example, strings such as `"abc"` are of type `xs:string`, numbers `56` may represent integers of type `xs:integer`, `56E3` is a representation of a double of type `xs:double`, booleans are of type `xs:boolean`, and names `math:sin` represent an element of type `QName` that contain some local part (`sin`) and optionally some namespace prefix (`math`). For these types XPath 3.0 functions are provided that manipulate their values, such as `fn:abs`, `fn:floor`, `math:exp`, and `math:sin` for numeric values, `fn:concat`, `fn:normalize-space`, and `fn:contains` for string values, and `fn:prefix-from-QName` and `fn:local-name-from-QName` for **QNames**.

4.2.2 Union Types

There exist union types as well in XPath 3.0. They are also derived by nonterminal `AtomicOrUnionType` (70) and denote the union of the denotation of two item types.

4.2.3 Node Types

The node types are derived by nonterminal `KindTest` (71).

```
(71) KindTest ::= DocumentTest | ElementTest | AttributeTest |
                  SchemaElementTest | SchemaAttributeTest | PITest |
                  CommentTest | TextTest | NamespaceNodeTest |
                  AnyKindTest
```

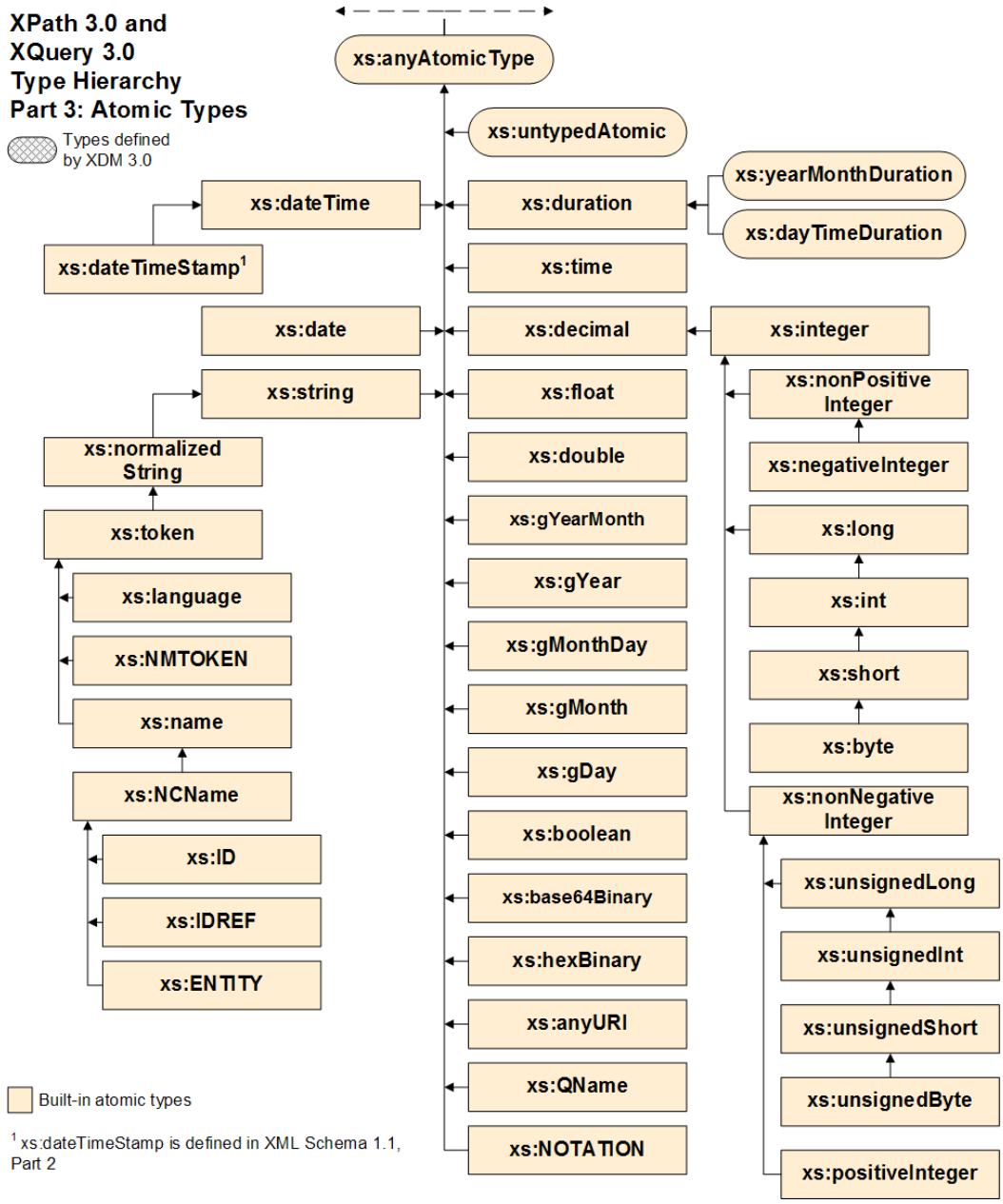


Figure 4.1: Excerpt of the XPath Data Model 3.0 specification about atomic types and their subtype relations.

Amongst the node types there are types `document-node()`, `element()`, `attribute()`, `text()`, `comment()`, and `processing-instruction()` which denote nodes of the respective kind, while `node()` denotes all nodes of all node kinds. Furthermore there are node types by nonterminal `ElementTest` (82), for example, that may also specify the label of an element node. For example, type `element(a)` denotes the

set of element nodes whose label is `a`. For nodes there exist built-in functions, such as `fn:name`, `fn:local-name`, `fn:path`, `fn:innermost`, and `fn:outermost` to name a few.

4.2.4 Function Types

Function types are produced by nonterminal `FunctionTest` (90). Type `function(*)`, for example, denotes any function. Function types may also contain the types of their signature, such as type

```
function(xs:integer, xs:integer) as xs:integer
```

which denotes all two-parameter functions of signature `xs:integer`×`xs:integer` → `xs:integer`. Built-in functions are provided, such as `math:sin` which return the sine of its argument. Note that functions may be of higher order, i.e. return a function as its result or take a function as one of its arguments, such as built-in functions `fn:for-each` and `fn:fold-left`. The signature of these functions however use sequence types, as we discuss next.

4.3 Sequence Types

Sequence types are build from **item** types. They optionally use one of the occurrence indicators "`?`", "`*`", or "`+`". **Sequence** types denote **sequences** that are ordered lists of **items** that are not nested. Furthermore a singleton sequence, i.e. a sequence containing exactly one item, is equivalent to the item it carries, for example, the double `56E3` = `(56E3)`.

```
(67) SequenceType       ::= ("empty-sequence" "(" ")") |
                            (ItemType OccurrenceIndicator?)
(68) OccurrenceIndicator::= "?" | "*" | "+"
```

There is the empty sequence type `empty-sequence()` which denotes the empty list. A **sequence** type that is an **item** type (no occurrence indicator is present) denotes any one-item sequence of an item that denoted by the **item** type. For example, the **item** type `xs:integer` is also a **sequence** type: `xs:integer` as an **item** type denotes the set of integers, while `xs:integer` as a **sequence** type denotes the set of one-item lists of integers. This is since singleton sequences are identified to the item they carry, making every item of XPath 3.0 a sequence.

An occurrence operator indicates the number of **items** in the **sequence**: "`?`" indicates zero or one items, "`*`" indicates zero or more items, and "`+`" stands for one or more items. A sequence type that is build from an **item** type with an occurrence operator "`?`", "`*`", or "`+`" denotes a list whose number of elements fits the occurrence operator, and whose elements are **items** of the respective **item** type.

For example the **sequence** type

```
xs:integer+
```

denotes a sequence of one or more integers, such as sequence

```
(1, 2, 3).
```

4.4 Subtyping

XPath 3.0 defines a subtype relation amongst item types and sequence types. The subtype relation of XPath is first defined for item types and then lifted to sequence types. We refer to the specification[1] of a complete description of the subtype relation and mention only a few here.

4.4.1 Item Types

Any node, function, or atomic type is a subtype of the general item type `item()`. Any function type is a subtype of the general function type `function(*)`, and any node type is a subtype of `node()`. For atomic types there are the subtypes indicated by arrows in Figure 4.1, for example

`xs:integer` is a subtype of `xs:decimal`.

4.4.2 Sequence Types

The subtype relation for item types is extended to a subtype relation on sequence types. This has to take into account all possible combinations of occurrence operators. For example, the sequence type `T?` for some item type `T`, is a subtype of sequence type `T'*` for some other item type `T'`, if `T` is a subtype of `T'`, e.g.

`xs:integer?` is a subtype of `xs:decimal*`

Note that the occurrence indicators may not exclude themselves, for example, a sequence type with occurrence indicator "`+`" cannot be a subtype of a sequence type with occurrence indicator "`?`", since the occurrence "`?`" does not match more than one item, and since "`+`" does not match zero items. Note that the empty sequence type "`()`" is a subtype of all sequence types with indicators "`?`" and "`*`", but not a subtype for those with "`+`" or without an indicator.

4.5 Dynamic Type Checking

Dynamic type checking is referred to in the XPath 3.0 specification as **Sequence Type Matching**[2]. Given some value it tests whether its type is **derived from** an expected sequence type.

[1] http://www.w3.org/TR/xpath-30/#id-sequencetype-subtype
[2] http://www.w3.org/TR/xpath-30/#id-sequencetype-matching

Function **derives-from** that is stated in the specification mainly describes the transitivity of the subtype relation. It can be defined as follows: Let $\mathtt{T}$, $\mathtt{T}'$, and $\mathtt{T}''$ be item types. Then $\mathtt{T}$ derives from itself. $\mathtt{T}$ derives from $\mathtt{T}'$, if $\mathtt{T}$ is a subtype of $\mathtt{T}'$. $\mathtt{T}$ derives from union type $\mathtt{T}' \cup \mathtt{T}''$ if either $\mathtt{T} = \mathtt{T}'$ or $\mathtt{T} = \mathtt{T}''$. Finally $\mathtt{T}$ derives from $\mathtt{T}''$ if $\mathtt{T}$ derives from some $\mathtt{T}'$, which in turn derives from $\mathtt{T}''$.

This allows to check whether a type *matches* some value, as follows. An item types T simply matches some value v, if the type of v is derived from T with respect to the above function *derives-from*. For sequences it can be defined as follows: the empty sequence type matches the empty sequence. An item type without an occurrence indicator matches a sequence if it contains just one item that is matched by the item type. An item type with an occurrence indicator matches a sequence if the number of elements in the sequence match the occurrence indicator and if the item type matches every item in the sequence.

Chapter 5

Full XPath 3.0

Contents

In this chapter we present most of the expressions allowed in XPath 3.0.

5.1 Grammar

5.1.1 Basic Expressions

```
(1)  XPath              ::= Expr
(2)  ParamList          ::= Param ("," Param)*
(3)  Param              ::= "$" EQName TypeDeclaration?
(4)  FunctionBody       ::= EnclosedExpr
(5)  EnclosedExpr       ::= "{" Expr "}"
(6)  Expr               ::= ExprSingle ("," ExprSingle)*
(7)  ExprSingle         ::= ForExpr | LetExpr | QuantifiedExpr |
                            IfExpr | OrExpr
(8)  ForExpr            ::= SimpleForClause "return" ExprSingle
(9)  SimpleForClause    ::= "for" SimpleForBinding
                            ("," SimpleForBinding)*
(10) SimpleForBinding   ::= "$" VarName "in" ExprSingle
(11) LetExpr            ::= SimpleLetClause "return" ExprSingle
(12) SimpleLetClause    ::= "let" SimpleLetBinding
                            ("," SimpleLetBinding)*
(13) SimpleLetBinding   ::= "$" VarName ":=" ExprSingle
(14) QuantifiedExpr     ::= ("some" | "every") "$" VarName "in"
                            ExprSingle ("," "$" VarName "in"
                            ExprSingle)* "satisfies" ExprSingle
(15) IfExpr             ::= "if" "(" Expr ")" "then" ExprSingle
                            "else" ExprSingle
(16) OrExpr             ::= AndExpr ( "or" AndExpr )*
(17) AndExpr            ::= ComparisonExpr ( "and" ComparisonExpr )*
(18) ComparisonExpr     ::= StringConcatExpr ( (ValueComp |
                            GeneralComp | NodeComp)
                            StringConcatExpr )?
(19) StringConcatExpr   ::= RangeExpr ( "||" RangeExpr )*
(20) RangeExpr          ::= AdditiveExpr ( "to" AdditiveExpr )?
(21) AdditiveExpr       ::= MultiplicativeExpr ( ("+" | "-")
                            MultiplicativeExpr )*
(22) MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" |
                            "mod") UnionExpr )*
(23) UnionExpr          ::= IntersectExceptExpr ( ("union" | "|")
                            IntersectExceptExpr )*
(24) IntersectExceptExpr::= InstanceofExpr ( ("intersect" |
                            "except") InstanceofExpr )*
(25) InstanceofExpr     ::= TreatExpr ( "instance" "of"
                            SequenceType )?
(26) TreatExpr          ::= CastableExpr ( "treat" "as"
                            SequenceType )?
```

```
(27) CastableExpr       ::= CastExpr ( "castable" "as" SingleType )?
(28) CastExpr           ::= UnaryExpr ( "cast" "as" SingleType )?
(29) UnaryExpr          ::= ("-" | "+")* ValueExpr
(30) ValueExpr          ::= SimpleMapExpr
(31) GeneralComp        ::= "=" | "!=" | "<" | "<=" | ">" | ">="
(32) ValueComp          ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
(33) NodeComp           ::= "is" | "<<" | ">>"
(34) SimpleMapExpr      ::= PathExpr ("!" PathExpr)*
```

5.1.2 Navigational Expressions

```
(35) PathExpr         ::= ("/" RelativePathExpr?)  | ("//"
                          RelativePathExpr) | RelativePathExpr
(36) RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
(37) StepExpr         ::= PostfixExpr | AxisStep
(38) AxisStep         ::= (ReverseStep | ForwardStep)
                          PredicateList
(39) ForwardStep      ::= (ForwardAxis NodeTest) |
                          AbbrevForwardStep
(40) ForwardAxis      ::= ("child" "::") | ("descendant" "::") |
                          ("attribute" "::") | ("self" "::") |
                          ("descendant-or-self" "::") |
                          ("following-sibling" "::") |
                          ("following" "::") | ("namespace" "::")
(41) AbbrevForwardStep::= "@"?  NodeTest
(42) ReverseStep      ::= (ReverseAxis NodeTest) |
                          AbbrevReverseStep
(43) ReverseAxis      ::= ("parent" "::") | ("ancestor" "::") |
                          ("preceding-sibling" "::") |
                          ("preceding" "::") |
                          ("ancestor-or-self" "::")
(44) AbbrevReverseStep::= ".."
(45) NodeTest         ::= KindTest | NameTest
(46) NameTest         ::= EQName | Wildcard
(47) Wildcard         ::= "*" | (NCName ":" "*") |
                          ("*" ":" NCName) |
                          (BracedURILiteral "*")
```

5.1.3 Postfix Expressions

```
(48) PostfixExpr        ::= PrimaryExpr (Predicate |
                            ArgumentList)*
(49) ArgumentList       ::= "(" (Argument ("," Argument)*)?  ")"
(50) PredicateList      ::= Predicate*
```

```
(51) Predicate           ::= "[" Expr "]"
(52) PrimaryExpr         ::= Literal | VarRef |
                             ParenthesizedExpr | ContextItemExpr |
                             FunctionCall | FunctionItemExpr
(53) Literal             ::= NumericLiteral | StringLiteral
(54) NumericLiteral      ::= IntegerLiteral | DecimalLiteral |
                             DoubleLiteral
(55) VarRef              ::= "$" VarName
(56) VarName             ::= EQName
(57) ParenthesizedExpr   ::= "(" Expr?  ")"
(58) ContextItemExpr     ::= "."
(59) FunctionCall        ::= EQName ArgumentList
(60) Argument            ::= ExprSingle | ArgumentPlaceholder
(61) ArgumentPlaceholder::= "?"
(62) FunctionItemExpr    ::= NamedFunctionRef | InlineFunctionExpr
(63) NamedFunctionRef    ::= EQName "#" IntegerLiteral
(64) InlineFunctionExpr  ::= "function" "(" ParamList?  ")" ("as"
                             SequenceType)?  FunctionBody
(65) SingleType          ::= SimpleTypeName "?"?
(66) TypeDeclaration     ::= "as" SequenceType
```

5.1.4 Terminal Expressions

```
(95)  IntegerLiteral   ::= Digits
(96)  DecimalLiteral   ::= ("." Digits) | (Digits "." [0-9]*)
(97)  DoubleLiteral    ::= (("." Digits) | (Digits ("."
                           [0-9]*)?))  [eE] [+-]?  Digits
(98)  StringLiteral    ::= ('"' (EscapeQuot | [^"])* '"') |
                           ("'" (EscapeApos | [^'])* "'")
(99)  URIQualifiedName::= BracedURILiteral NCName
(100) BracedURILiteral::= "Q" "{" [^{}]* "}"
(101) EscapeQuot       ::= '""'
(102) EscapeApos       ::= "''"
(103) Comment          ::= "(:" (CommentContents | Comment)* ":)"
(104) QName            ::= [http://www.w3.org/TR/REC-xml-names
                           /#NT-QName]
(105) NCName           ::= [http://www.w3.org/TR/REC-xml-names
                           /#NT-NCName]
(106) Char             ::= [http://www.w3.org/TR/REC-xml#NT-Char]
(107) Digits           ::= [0-9]+
(108) CommentContents  ::= (Char+ - (Char* ('(:'  | ':)')  Char*))
```

Figure 5.1: The parse tree of XPath query `descendant::tweet`.

5.1.5 Parse Tree

We present the parse tree of XPath query `descendant::tweet` for the XPath 3.0 grammar in Figure 5.1. This is the parse tree using the original XPath 3.0 grammar, in contrast to the parse tree of Figure 3.1 for the same query, but for the navigational XPath grammar given in Section 3.2.2.

5.2 Values

5.2.1 Strings and Numbers

There are **literal** expressions, created by nonterminal `Literal` (53), which are expressions that either return a string or a numeric value. They are syntactic representations of a string or a numeric value.

A numeric expression without "`.`", "`e`", or "`E`" is an integer of type `xs:integer`. It is of type `xs:decimal` if it only has the dot "`.`", and otherwise it is a value of type `xs:double`. For example, expression `84` is a decimal value of type `xs:decimal`. They can also be constructed using constructor functions that specify the atomic type. For example, `xs:integer(84)` creates an integer value of type `xs:integer`. Furthermore, we would like to mention `xs:float("NaN")` that returns special floating-point value "Not a Number", and `xs:double("INF")` that return a special value for "positive infinity".

String expressions are surrounded by double quotes and denote the string of characters within the quotes. For example expression `"8.4"` denotes a string containing the characters "`8`", "`.`", and "`4`". Strings can be concatenated via the string concatenation expression by `StringConcatExpr` (19) that uses symbol "`||`" to denote the concat function `fn:concat`. For example, query `"8." || "4"` returns string `"8.4"`.

5.2.2 Booleans

The boolean values *true* and *false* are returned by XPath 3.0 functions `fn:true` and `fn:false`, respectively. Boolean values in XPath 3.0 are computed by conditionals, logical connectives, filters, quantifications, and comparisons. However also other data types than booleans are permitted as test expressions, such as node sequences, strings, and numeric values. For example, query

```
if //a then //b else ()
```

evaluates to the sequence of `b`-descendants, if there is an `a`-descendant, and otherwise it evaluates to the empty sequence.

How to evaluate to boolean values in general is defined in XPath by the **effective boolean value**, which is computed by function `fn:boolean`. This function is computed implicitly for all expressions requiring boolean values. The above example is therefore equivalent to

```
if fn:boolean(//a) then //b else ()
```

For node sequences, functions `fn:boolean` returns *true*, if the sequence contains at least one node. For string types, the function returns *true*, if the value is not the empty string. For numeric values that are not equal to 0 and not the constant *NaN* (not a number), *true* is returned and for boolean values the value itself is returned.

5.2.3 Functions

Function expressions. Functions can be referenced using EQNames, such as built-in function `math:sin`. The whole list of built-in functions is given in [Kay 2014].

Functions may be also referenced by an expression `name#arity` that uniquely identifies a function via its name and arity such as `concat#2`, which returns the concat function with two arguments.

Functions can also be defined inline within the XPath expression, such as

```
function($a as xs:double, $b as xs:double) as xs:double {$a * $b},
```

that returns a function with two arguments of type `xs:double` computing their product. Inline function expressions specify the sequence types of the parameters of the function, the sequence type of the result, and the body of the function (the part within the curly brackets). If a sequence type is not declared, the default type is `item()*`.

Last but not least in XPath 3.0 there are function call expressions that identify a function via an EQName, and which is called with a list of arguments. If some of the arguments are argument placeholder expressions `?`, it is considered to be a partial function application that returns a function that is obtained via currying. For example the partial function application `fn:concat("a", ?, "c")` results in function $\lambda y.$`fn:concat("a",` y`, "c")`.

Atomization and Coercion. The evaluation of a function requires to convert the value of its arguments to their expected type, which is specified by the *function conversion rules*[1]. If one of its arguments types is a function type, then function coercion is applied to each function of the argument sequence, see next paragraph. Otherwise the expected argument type is a sequence type of non-function items. Atomization is applied to every item in that sequence, resulting in a sequence of atomic values. Those values are then converted to the expected non-function item type, if possible.

The specification defines *function coercion*[2], which converts a function that is passed as argument to another function, in order to delay the checking of argument and return types until the invocation of the function. The function is converted into a new function that has the expected function type. When the new function is invoked, function conversion rules are applied to the function arguments, with which the original function is called. This result is finally converted to the expected result type of the function, applying once more the function conversion rules.

5.2.4 Sequences

Construction. The empty sequence is constructed with query `()`. In general sequences are constructed using the comma as in `(((2)), (3.0, "4"), ())` which is subject to sequence flattening.

[1] http://www.w3.org/TR/xpath-30/#id-function-conversion-rules
[2] http://www.w3.org/TR/xpath-30/#id-function-coercion

Flattening. Sequences are not nested, in that a sequence can not appear as an element of another sequence. When combining sequences and items, an implicit flattening occurs, e.g.

```
(((2)), (3.0, "4"), ()),
```

where `()` is the empty sequence, is flattened to the sequence

```
(2, 3.0, "4").
```

Decomposition. Sequence are decomposed via for expressions by nonterminal `ForExpr` (8), such as

```
for $x in (1, 33, 555) return ($x + 100, $x)
```

which evaluates to sequence `(101,1,133,33,655,555)`. The expression following the `return` keyword is evaluated for every item `$x` in the sequence by the expression before the `return` keyword and in that order. Result sequences are appended. Moreover, functions that operate on sequences exist as built-in functions, such as `fn:head`, `fn:tail`, `fn:remove`, and `reverse`.

Atomization. The evaluation of arithmetic expressions and comparison expressions, relies on *atomization*, which atomizes a sequence of atomic values, nodes, and functions, to obtain a sequence of atomic values only. This is computed by function `fn:data`, which retains atomic values, raises an error when encountering functions, and which computes the typed value of node items. The typed value is the data content of text, comment, and processing-instruction nodes, which corresponds to the attribute value for attribute nodes, and to the concatenation of data content of text node descendants for document and element nodes. For attribute and element nodes the result string is then cast to the schema-type stored as data value property of attribute and elements nodes.

For example, if the value of the argument to function `fn:data` is already atomic, as in query `fn:(12.5)`, then the atomic value is returned. Otherwise for the small document "`<a>Hello <b>there</b>.</a>`" query

```
fn:data(//a)
```

returns the untyped atomic value `"Hello there."`: The function gets a list of `a`-nodes. Since nodes are not atomic, the nodes are atomized to their typed value.

Type Checking. An example XPath 3.0 query that performs sequence type matching directly is expression `P instance of P'` which returns true whenever the type of the result by `P` matches the type by `P'`. For example

```
(1, 2, 3) instance of xs:integer+
```

evaluates to true.

5.2.5 Conversions

XPath 3.0 provides cast expressions that allow to cast values to a specific type, for which the specification gives a vast list of conversion rules[3] that we will not discuss in any detail. An example is the following cast expression that casts value `2` as a double value:

```
2 cast as xs:double
```

For the purpose of comparing values using comparison expressions, see Section 5.6, one may require to find the *least common type* of two types, to which respective values are converted. For example the least common type of an `xs:integer` and a `xs:float` value is `xs:float`.

5.3 Positions

XPath makes vastly use of *positions*, which is the facility to test filters and to evaluate path expressions according to some position in a sequence. It means to reason about an item in a list, its position, and the size of the list. This information has to be accessible when looking at just one item in the sequence, thus one would require lists of tuples with this information when implementing it. However this is not supported directly within XPath 3.0, since sequences are flat.

For path expression using the path operator in expressions `P/P'`, or the simple map operator (see Section 5.4.2) in expressions `P!P'`, or when testing predicates `P[P']`, XPath defines a *dynamic context* for the evaluation of sub-expressions `P'`, which contains information about the evaluation of `P`: The *context item*, referred to by XPath expression "`.`", is the item which is currently being processed, whose initial value is to be set by an implementation. For path expressions and filters, the context item for `P'` is the item of sequence `P` that is currently being processed. The *context position*, referred to by function `fn:position()`, is the position of the context item within the sequence. Finally the *context size* is the number of items in the sequence, and referred to by function `fn:last()`.

For example consider query

```
(1 to 30)[.  mod 3 eq 0],
```

where `1 to 30` is a range expression that evaluates to the sequence of consecutive integers 1 until 30. Each of those integers eventually becomes the context item, for which the filter `.  mode 3 eq 0` evaluates to true if and only if it is dividable by 3. The result sequence is therefore $3, 6, 9, \ldots, 27, 30$. The context size for the filter is 30 and the context item happens to be the context position. A query referring to context position and size is

```
//a[fn:position()=fn:last()] = //a[fn:last()]
```

[3] http://www.w3.org/TR/xpath-30/#id-cast

which returns the last a-node in document order: The number of a-nodes in the sequence is fn:last(), while the positions of the a-nodes range from 1 until fn:last() such that the equality becomes true only for the last a-node. Last but not least, we mention a query that uses the path operator /, and which returns a sequence of consecutive integers: one for every a-node, starting with 1:

```
//a/fn:position()
```

5.4 Navigation

5.4.1 Path Expressions

Path expressions are produced by nonterminal PathExpr (35), which we discussed in detail in Chapter 3. Path expressions according to the full XPath 3.0 grammar may also be **postfix** expressions, i.e. expressions by nonterminal PostfixExpr (48), that produce **literal** expressions, variable reference expressions, function calls, and function expressions.

5.4.2 Map Operator

One of the new features of XPath 3.0 are expressions by nonterminal SimpleMapExpr (34) that use the mapping operator "!", which in contrast to the path operator operates on **sequences** of **items**, not only on **sequences** of nodes or non-node. It does not perform duplicate elimination, but only appends **sequences** by P' in the order of **items** of the **sequence** by P for mapping expressions P!P'. For example, query

```
(1,3,5) ! (. + 10)
```

returns the sequence (11,13,15): Subexpression (. + 10) is evaluated for every item in (1,3,5) where the context item "." takes value 1, 3, and 5. Its result sequence therefore contains items 11, 13, and 15 in that order. However when XPath 3.0 is used in host language XQuery 3.0, whose ordering mode is set to *unordered*, then the result **sequence** may be returned in an implement-dependent order, see also Section 5.7.2.

Interestingly, the path expressions P/P' that use the path operator "/" can be expressed by the mapping operator which uses operations on sets of nodes:

```
P/P' =
 let $R:= P!P' return
  if (every $r in $R satisfies $r instance of node()) then
    ($R union ())
  else if (every $r in $R satisfies not($r instance of node())) then
    $R
  else error()
```

First, expression P!P' is evaluated. Then the result sequence is analyzed for whether only nodes are contained. If so, the union operator union is applied to the result

sequence with the empty one: Duplicates are eliminated (see Section 5.7.1). Otherwise if there exist an item in the sequence that is not a node, all items must be non-nodes and the sequence is returned, otherwise an error is thrown.

5.5 First-Order Connectives

XPath 3.0 provides first-order logic constructs, such as the logical connectives: conjunctions, disjunctions, negation, and quantifications.

Logical connectives like function `fn:not` and disjunction `or` perform implicit calls to `fn:boolean` for their operands as in the following query. The function computes the **effective boolean value** as discussed in Sectionchap3:sec:booleanvalues.

```
//book[fn:not(title) or fn:not(author)]
```

that selects all `book`-nodes, which either have no `title` or no `author`.

XPath provides existential quantification `some $y in P satisfies P'` that evaluate to *true*, if the effective boolean value of at least one evaluation of `P'` with variable $y bound to an element of the sequence by `P` is *true*. The quantification `every $y in P satisfies P'` requires function `fn:boolean` to become *true* for all evaluations of `P'` as in query

```
every $y in (1, "b", fn:true()) satisfies $y
```

which evaluates to *true*, since the `fn:boolean(1)` = *true*, `fn:boolean("b")` = *true* and since the boolean constant `fn:true()` evaluates to *true*.

5.5.1 Arithmetics

Arithmetic operators are provided for addition "`+`", subtraction "`-`", multiplication "`*`", division "`div`" and "`idiv`", modulus "`mod`", and for unary expressions "`+`" and "`-`".

Arithmetic expressions `P op P'` are evaluated as follows, where `op` is one of the mentioned arithmetic operators. `P` and `P'` are evaluated to sequences, that undergo atomization. Then when the sequences contain exactly one atomic value, whose types are of *valid combination*[4] for operator `op`, then the result is computed. Certain errors are raised if some of the mentioned condition for the computation of the arithmetic expressions are not met, or from division by 0 for example. The division of `-3 div 2` for example returns a decimal value of `-1.5`, while the *integer division* `-3 idiv 2` returns integer `-1`.

5.6 Data Comparisons

Two values can be compared using one of XPath 3.0 comparison expressions: value comparisons, node comparisons, and sequence comparisons.

[4] http://www.w3.org/TR/xpath-30/#mapping

5.6.1 Atomic

Atomic value comparison expressions, a.k.a. **value comparisons** in XPath, use comparison operators "`eq`" (equals), "`ne`" (not equals), "`lt`" (less than), "`le`" (less than or equals), "`gt`" (greater than), and "`ge`" (greater than or equals). These expressions are evaluated similar to arithmetic expressions, in that sequences of the operators are atomized, and in that they should contain only a single item. The resulting two values, however, are first converted to their least common type before they are compared, assuming that they are valid combinations. An example is XPath query

```
//book[pages lt 200]
```

which selects all `book`-descendants of less than `200` `pages`.

5.6.2 Nodes

Node comparison expressions use operator "`is`", which checks node identity, and operators "`<<`" and "`>>`", which checks whether a node appears before or after another node in document order, respectively. No atomization occurs, but sequences must evaluate to single-node sequences. Query

```
//book[isbn="34668966"] is /library/book[title="Data"]
```

returns *true*, if the `book`-descendant with `isbn`-number `"34668966"` is the same node as a certain `book`-node, whose `title` is `"Data"`.

5.6.3 Sequences

For sequence comparisons, a.k.a. **general comparisons** in XPath, two sequences are being joined together in order to find respective elements that compare to true with respect to the sequence comparison operators "`=`" (equals), "`!=`" (not equals), "`<`" (less than), "`<=`" (less than or equals), "`>`" (greater than), and "`>=`" (greater than or equals). Atomization is applied to each operand of the comparison to obtain sequences of atomic values. In essence the result of the comparison is true, if there exists a value from the first sequence and another value from the second one, such that they compare to true with respect to the comparison operator. Behind the scene a variety of type conversion are carried out in order to obtain comparable types for the atomic values that are being compared. For example query

```
//book[author = "Abiteboul"]
```

selects all `book`-descendant, who have an `author`-child whose concatenation of data content of its text node descendants is equal to `"Abiteboul"`.

5.7 Ordered Sets

The concept of sets or sets of nodes is present in XPath 1.0. There the type *node-set* is used, which is considered to be an unordered collection of nodes without

duplicates. However when serializing or enumerating a set of items or a set of nodes, one always needs to fix an order. Therefore in XPath 3.0 the concept of ordered sets moved forward. It is represented as a sequence without duplicates.

5.7.1 Duplicate-free Sequences

The selection of nodes via navigation, as discussed in Section 3.3, produces a sequence of nodes without duplicates, that if not specified otherwise (see below in Section 5.7.2), is sorted in document order.

Also basic operations on sets of nodes are provided: unions, intersections, and complements. They take two sequences of nodes and compute the union, intersection, and set difference, respectively. All operators eliminate duplicates and subsequently sort the nodes in document order, computing thus an ordered set of nodes. For example XPath query

```
//a//c intersect //b//c
```

selects all `c`-nodes that are descendants of both an `a`-descendant and a `b`-descendant of the root. Query `//a//c union //b//c` selects all `c`-nodes that are descendants of either an `a`- or a `b`-descendant of the root. Last but not least query `//a//c except //b//c` selects all `c`-nodes that are descendants of `a`-descendants of the root, but which are no descendants of `b`-root descendants.

Furthermore there is a function `fn:distinct-values` which returns an ordered set. It takes a sequence of items and returns a new duplicate-free sequence that removes all but one of a set of values that are equal to one another. The sequence may be returned in an implementation dependent order. We note that this function heavily depends on the implementation of XPath's equality operator `eq`.

5.7.2 Computing Sets when possible

For efficiency reasons one is interested in returning sequences in an arbitrary implementation dependent order. This can be obtained within XPath 3.0 by applying `fn:unordered`. However it relaxes the ordering only for the sequence that is its immediate operand. For query

```
fn:unordered(for $y in (1, 2, 3) return $y + 10)
```

any permutation of $(11, 12, 13)$ is an admissible answer. However the sequences are still concatenated in the order of sequence `(1, 2, 3)`, and this cannot be undone by applying function `fn:unordered` to the concatenation.

Within the host language of XQuery the order in which sequences are produced can be relaxed in XPath 3.0 expressions according to the ordering mode of XQuery. This "order indifference" in XQuery expressions was already discussed in [Grust 2007] and exploited in a purely algebraic fashion.

XQuery defines the *ordering mode* $\in \{\texttt{ordered}, \texttt{unordered}\}$, which has the two values `ordered` and `unordered` that affect the order in which sequences of

expressions and all nested expressions thereof are returned. XQuery expressions `ordered{P}` and `unordered{P}` determine the ordering mode of expressions `P` and its subexpressions. The ordering mode affects path expressions, node-set operations, for expressions, and allows freedom even with position functions (Section 5.3) that depend on the order of items in a sequence. For example the XQuery

```
unordered{//a/b//c}
```

would not have to sort the `a`-descendants found by subexpression `//a` into document order, before applying the path operator `/` to find `b`-children and `c`-descendants thereof.

5.8 Error Handling

5.8.1 Kinds of Errors

Errors may be raised in XPath 3.0 during static analysis or during the evaluation of the query. Errors raised during the static analysis phase, that are no type errors, are called *static errors* as with query

```
30div 5
```

which results in a syntax error. *Type errors* occur when some type does not match another expected type. They may occur during static analysis or during the evaluation phase. Finally, *dynamic errors* are mostly those errors that are detected during the evaluation phase. Consider XPath query

```
(5 div $y) + xs:decimal($z)
```

which may result in a dynamic error, when `$y` evaluates to 0 or when `$z` evaluates to string `"abc"`, for example. An implementation however may choose which of the two errors is raised by the additive expression and if some operand of some expression raises an error, then it may always decide to evaluate the other operands as well in order to detect whether errors occur there as well.

5.8.2 Try and Catch Expressions

Try and catch expressions do not exist in XPath 3.0 as built-in constructs, although they exist in XQuery 3.0. However, they exist in XPath 3.0 implicitly. There is an expression in XPath that tests whether a given value is castable into a certain type:

> "The expression `E castable as T` returns `true` if the result of evaluating `E` can be successfully cast into the target type `T` by using a `cast` expression; otherwise it returns false."

The castable-as expression thus performs a real cast (Section 5.2.5), and it catches any dynamic errors that it may throw.

5.9 Documents

XPath 3.0 provides a collection of functions that access input documents and external resources, and other functions that allow to create Xml data trees on the fly.

There is function `fn:doc` that identifies an Xml data tree within a sequence of available documents using a string that contains a Uri and it returns its root node, which is a document node. The query

```
fn:doc("library.xml")/library//books
```

selects all `book`-descendants of `library`-children of the root within the Xml data tree that is associated with URI `"library.xml"`.

Access to external resources, for example a text file, is granted by function `fn:unparsed-text` that takes a string that contains a Uri as argument and optionally an encoding and that returns a string that represents the content of the file associated with the Uri. An example query that checks whether a string is contained in a text file is:

```
fn:contains(fn:unparsed-text(passwords.txt),"#$df&@Wdf")
```

Last but not least we mention function `fn:collection` that returns a sequence of nodes of Xml data trees. The function associates the Uri `"http://example.org"` in query

```
fn:collection("http://example.org")//people
```

with a collection of nodes.

5.10 Regular Extension of XPath 3.0

By *Regular* XPath *3.0* we mean an extension of XPath 3.0 with regular axis `P*`, which repeats steps for some path expression `P`. This is a pure syntactic extension, since regular axis can be expressed by *recursive* functions in XPath 3.0 as we show below. Syntactically it can be integrated into the grammar of XPath 3.0 by adding a pattern `"(" RelativePathExpr ")*"` to nonterminal `PathExpr` (35), obtaining

(35') `PathExpr::= ("/" RelativePathExpr?) | ("//" RelativePathExpr) | RelativePathExpr | "(" RelativePathExpr ")*"`

Its semantics can be understood as follows. The context item, i.e. the item at which `P*` is evaluated, must be a node, and `P` must always evaluate to a sequence of nodes. Then `P*` evaluates to a sequence of nodes, which contains the context item and which contains recursively all nodes that an evaluation of `P` yields from a node that is contained in the sequence by `P*`.

However, the extension with regular axis to the grammar is not necessary, although we keep it for simplicity, since regular axis `P*` can be expressed in XPath

```
P* =df
 let $closureP :=
       function($nodes as node()*,
                $closure as function(node()*,function()) as node()*)
         {let $newnodes := $nodes union ($nodes/P)
          return if fn:exists($newnodes except $nodes) then
                   $closure($newnodes,$closure)
                 else
                   $newnodes
         },
     $regaxis :=
       function($nodes as node()*) as node()*
         {$closureP($nodes,$closureP)}
 return $regaxis(.)
```

Figure 5.2: Regular axis `P*` by let expressions and recursive function definitions.

3.0 already as we show in Figure 5.2 using let expressions and *recursive* function definitions. There the outer let expression defines two variables `$closureP` and `$regaxis`, which are assigned to (unnamed) functions. Variable `$regaxis` is assigned to a function that takes one argument, which is a list of nodes. It is invoked with the context node ".", meaning the list containing only the context node. It then invokes variable `$closureP`, which is assigned a function with two parameters: the first being a list of nodes and the second parameter is function `$closure` also with two parameters. The function proceeds by computing the union with all nodes reached from nodes in `$nodes` by path expression `P`, obtaining possibly a bigger set of nodes in `$newnodes`. If the new set of nodes is the same as the start set of nodes in `$nodes`, then it is returned, otherwise the function goes into recursion with the new set of nodes in `$newnodes`.

For example, consider the XML data tree that represents a collection of tweets by TWITTER in Figure 1.4, where each `tweet`-node has children `id`, `reply`, `mess`, and `user`. Then the regular XPATH query

```
(let $y := .  return following-sibling::tweet[reply = $y/id])*
```

when started at the second `tweet`-node, selects recursively all `tweet`-following-sibling nodes that reply to it. They are marked in bold in the Figure 1.4.

5.11 XPathMark Benchmark

A common XPATH benchmark is the XPATHMARK benchmark[5] which is the revised version in contrast to the benchmark of the original paper [Franceschet 2005]. It

[5] http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html

contains queries about XMark documents that contain a table of bids and a table of bidders. Some of the queries are parameterized by an integer $i \geq 1$, so that these queries can be scaled in size. Notice that the XPathMark bechmark was designed to test XPath 1.0 queries only, which are valid XPath 3.0 queries however. We give an overview in the following about the types of queries in the benchmark, while we refer to Appendix A for the full benchmark query collection, of which we test only a subset in our experiments.

The benchmark contains six categories A–F of queries: downward (A), axis (B), comparison (C), aggregation (D), position (E), and closure queries (F). Downward queries in the benchmark use child and descendant axes, such as query

A3: `/site/closed_auctions/closed_auction//keyword`

or they contain filters with conjunctions or disjunctions, such

A7: `/site/people/person[phone or homepage]/name`

The axis queries in the benchmark use in addition to the downward axes, also backward axes, attribute axes and other forward axes. Some of the queries are parametric, such as

B12(i): `//item(/@id/..)`^{i}`/name`

The comparison queries in the benchmark perform mostly data joins, such as query

C2:
```
/site/open_auctions/open_auction[bidder/increase = current]
/interval
```

The benchmark contains various aggregation queries that allow to count items, or to sum up values:

D3:
```
/site/open_auctions/open_auction
[sum(bidder/increase) > 10 * initial]/interval
```

Positional queries of the benchmark contains positions as in query

E3: `//keyword/ancestor::listitem[1]/text/keyword`

Finally, there are some queries, that use some closure function, that repeatedly perform axes steps, similar to regular axes.

Chapter 6

λXP

Contents

In this chapter we introduce λXP, a typed first-order logical language that can query data trees. The language contains constructs for path navigation and supports *higher-order* functions and *recursion*. It relies on a mode system that distinguishes logical subformulas for which solutions must be searched allowing for efficient implementations. λXP is then used as a core language for implementing evaluators for XPath 3.0 in the next chapter.

6.1 Types and Values

We define a typed language for values including tree nodes, tuples, sets, lists, and higher-order types with unions.

6.1.1 Typing Parameters

The type language is parameterized by a set of atomic types, their subtyping relationships and their interpretation.

Definition 5. *The* type parameters *for sequences of data trees are a tuple $\mathcal{A}$ with the following components:*

- Σ *is a finite ranked set, the node constructors of the data trees.*
- Δ *is a finite set, the alphabet of the data values of the data trees.*
- D *is a set of elements,*
- $\mathcal{T}_{atom}$ *is a finite set whose elements are the atomic types to be interpreted over $D \uplus nodes(s)$ for any sequence s of data trees. They must contain:*

$$\{\mathbf{node}, \mathbf{char}, \mathbf{string}, \mathbf{none}, \mathbf{int}, \mathbf{float}, \mathbf{bool}, \mathbf{lab}\} \subseteq \mathcal{T}_{atom}$$

- *For any sequence s of data trees and any atomic type $T \in \mathcal{T}_{atom}$, there is a domain of elements inhibiting this type $[\![T]\!]_s \subseteq D \uplus nodes(s)$.*
- *For any sequence s of data trees, $\leq_D$ is a partial order on D which is total when restricted to elements of equality types $\mathcal{T}_{eq}$ defined below.*

6.1.2 Atomic Types

The set of atomic types must contain the type **node** for nodes of trees in the sequence, the type **char** of string characters in Δ, the type **string** of strings over Δ, the empty type **none**, the two numeric types **int** for integers and **float** for floats, the Boolean type **bool**, and the type **lab** of node constructors in Σ. These types have the following domains for any sequence of trees $s \in (\mathcal{T}_{\Sigma,\Delta})^*$:

$$\begin{aligned}
[\![\mathbf{node}]\!]_s &= \text{nodes}(s) \\
[\![\mathbf{char}]\!]_s &= \Delta \\
[\![\mathbf{string}]\!]_s &= \Delta^* \\
[\![\mathbf{none}]\!]_s &= \emptyset \\
[\![\mathbf{int}]\!]_s &= \mathit{Int} \\
[\![\mathbf{float}]\!]_s &= \mathit{Float} \\
[\![\mathbf{bool}]\!]_s &= \mathit{Bool} \\
[\![\mathbf{lab}]\!]_s &= \Sigma
\end{aligned}$$

We assume that nodes of s are identified by values that are different from all others, even though one may think of node identifiers as numbers. On atomic types we define a subtype relation $\preceq_{atom}$ semantically so that for all $T_1, T_2 \in \mathcal{T}_{atom}$:

$$T_1 \preceq_{atom} T_2 \text{ iff } \forall s.\ [\![T_1]\!]_s \subseteq [\![T_2]\!]_s$$

In particular, we have **int** $\preceq_{atom}$ **float** and **none** $\preceq_{atom}$ T for all $T \in \mathcal{T}_{atom}$. Furthermore, since we assumed that node identifiers are different from all other values, we have **node** cannot be subtype or supertype of any of other atomic type enumerated above. However, there may be some other atomic types, say for instance $\mathbf{node}_\sigma$ for all nodes labeled by $\sigma \in \Sigma$ and this type is a subtype of **node**.

6.1.3 Types

A type is a term build from atomic types according to the following abstract syntax:

$$T ::= A \mid T_1 \times \ldots \times T_n \mid [T] \mid \{T\} \mid T_1 \cup T_2 \mid T_1 \to T_2$$

Besides atomic types $A \in \mathcal{T}_{atom}$, there are types for n-tuples $T_1 \times \ldots \times T_n$, lists $[T]$, sets $\{T\}$, unions $T_1 \cup T_2$, and functions $T_1 \to T_2$, where T, T_1, T_2 are types again. The tuple type were $n = 0$ is denoted by **unit**. We can also define the type of numerals:

$$\mathbf{num} =_{\text{def}} \mathbf{int} \cup \mathbf{float}$$

To any type T and sequence s of trees we assign a value $[\![T]\!]_s$ by extending the definition for atomic types as follows:

$$\begin{aligned}
&[\![T_1 \times \ldots \times T_n]\!]_s = [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s \\
&[\![\{T\}]\!]_s = \{S \mid S \subseteq [\![T]\!]_s \text{ finite}\} \\
&[\![[T]]\!]_s = ([\![T]\!]_s)^* \\
&[\![T_1 \cup T_2]\!]_s = [\![T_1]\!]_s \cup [\![T_2]\!]_s \\
&[\![T_1 \to T_2]\!]_s = [\![T_1]\!]_s \to ([\![T_2]\!]_s \cup \{\bot\})
\end{aligned}$$

Note that the type **string** denotes the set of all character strings in Δ^*, while the denotations of **num** and **float** are both the set of floats, since **int** $\preceq$ **float**.

The function type $T_1 \to T_2$ is interpreted as the set of all partial functions $[\![T_1]\!]_s \to ([\![T_2]\!]_s \cup \{\bot\})$. The symbol $\bot$ is returned for all the values of $[\![T_1]\!]_s$ where the function is undefined. This is needed, for instance, to treat division by zero in a proper manner, but also for the treatment of non-termination of function evaluation.

6.1.4 Subtyping

We extend the subtype relation $\preceq_{atom}$ to a subtype relation on types $\preceq$ by means of the rules in Figure 6.1. We next note that subtyping entails set inclusion on the denotations of the types.

Lemma 2. *If* $T_1 \preceq T_2$ *then* $[\![T_1]\!]_s \subseteq [\![T_2]\!]_s$.

Proof. By induction of judgements of $T_1 \preceq T_2$. In the base case of $T_1 \preceq_{atom} T_2$ it holds by semantic definition of $\preceq_{atom}$. □

$$\frac{T_1 \preceq_{atom} T_2}{T_1 \preceq T_2} \qquad \frac{\mathit{true}}{\mathbf{none} \preceq T} \qquad \frac{\forall 1 \leq i \leq n:\ T_i \preceq T_i'}{T_1 \times \ldots \times T_n \preceq T_1' \times \ldots \times T_n'}$$

$$\frac{1 \leq i \leq n \qquad T_i \preceq T_i' \cup T_i''}{T_1 \times \ldots \times T_n \preceq \begin{array}{l}(T_1 \times \ldots \times T_i' \times \ldots \times T_n) \\ \cup \quad (T_1 \times \ldots \times T_i'' \times \ldots \times T_n)\end{array}}$$

$$\frac{T_1 \preceq T_2}{[T_1] \preceq [T_2]} \qquad \frac{T_1 \preceq T_2}{\{T_1\} \preceq \{T_2\}} \qquad \frac{\mathit{true}}{T_1 \preceq T_1 \cup T_2} \qquad \frac{T_1 \preceq T \qquad T_2 \preceq T}{T_1 \cup T_2 \preceq T}$$

$$\frac{T_1 \preceq T_2}{T_1 \cup T_2 \preceq T_2} \qquad \frac{T_1' \preceq T_1 \quad T_2 \preceq T_2'}{T_1 \to T_2 \preceq T_1' \to T_2'}$$

Figure 6.1: Subtyping.

6.1.5 Equality Types

For defining the semantics of our language, we will distinguish types whose elements can be compared for equality and ordered totally from the others. In particular, it is not clear how to test functions for equality, so we rule we exclude function type from the set of equality types.

Definition 6. *The set $\mathcal{T}_{eq}$ of* equality types *is the least set that contains all atomic types, and for all equality types $T, T_1, \ldots, T_n \in \mathcal{T}_{eq}$, the tuple type $T_1 \times \ldots \times T_n$, the list type $[T]$, and the union type $T_1 \cup T_2$.*

We assume that the partial order $\leq_D$ is total on all elements of equality types, i.e, on $D^{eq} = \cup_{T \in \mathcal{T}_{eq}} [\![T]\!]_s$. Thereby we obtain an equality relation on all elements of equality types:

$$=_{eq} \quad = \quad (D^{eq} \times D^{eq}) \cap \leq_D \cap \geq_D .$$

In practice the order $\leq_D$ is obtained as follows. We can assume that the values of atomic types are totally ordered, so this order will be subsumed by $\leq_D$. The order for comparing tuples $\leq_{T_1 \times \ldots \times T_n}$ is the lexicographic order on $[\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s$. The order for lists is the lexicographic order on $([\![T]\!]_s)^*$. Furthermore, we can have atomic types except strings before tuples and lists, while strings are to be identified with lists of characters.

We next show that subtypes of equality types are equality types too.

Lemma 3. *Subtypes of equality types are equality types.*

Proof. By induction of judgements $T_1 \preceq T_2$. Atomic subtypes are equality types by definition. For subtypes of types of n-tuples, the lemma hold by induction, and similarly for subtypes of list types. Subtypes T_1 or T_2 of union type $T_1 \cup T_2 \in \mathcal{T}_{eq}$ are equality types by definition. For subtyping rule $\frac{T_1 \preceq T \qquad T_2 \preceq T}{T_1 \cup T_2 \preceq T}$ it follows by induction that T_1 and T_2 are equality type, since $T_1 \preceq T$ and $T_2 \preceq T$ for equality type T. It follows that subtype $T_1 \cup T_2$ of T is an equality type by definition. For rule $\frac{T_1 \preceq T_2}{T_1 \cup T_2 \preceq T_2}$ it follows by induction that T_1 is an equality type since $T_1 \preceq T_2$

for equality type T_2. By definition also $T_1 \cup T_2$ is an equality type. Function and set types are ruled out, since we assume equality types. □

6.1.6 Admissible Types

Definition 7. *A type is* admissible *if for any of its subterms* $\{T\}$ *the type* T *is an equality type.*

Lemma 4. *Subtypes of admissible types are admissible.*

Proof. By induction of judgements $T_1 \preceq T_2$ it follows for all subtyping rules, except rule $\dfrac{T_1 \preceq T_2}{\{T_1\} \preceq \{T_2\}}$, that the inferred subtype is admissible, according to the definition of admissibility. For rule $\dfrac{T_1 \preceq T_2}{\{T_1\} \preceq \{T_2\}}$, we know that for admissible type $\{T_2\}$ the type T_2 is an equality type. By Lemma 3 the subtype T_1 of T_2 is an equality type. By definition we have that subtype $\{T_1\}$ of $\{T_2\}$ is an admissible. □

Therefore, we restrict ourselves to admissible types, for which the elements of sets have an equality type. This is particularly relevant for implementing finite sets, as for instance by ordered lists, so that one can remove duplicates efficiently.

Lemma 5. *For any two equality types* T_1, T_2*, the restrictions of* $\leq_{T_1}$ *and* $\leq_{T_2}$ *to* $[\![T_1]\!]_s \cap [\![T_2]\!]_s$ *coincide.*

As a consequence, if $T_1 \preceq T_2$ then $[\![T_1]\!]_s \subseteq [\![T_2]\!]_s$ by Lemma 2, so that $\leq_{T_1}$ is the restriction of $\leq_{T_2}$ to $[\![T_1]\!]_s$.

Proof. The orders $\leq_{T_1}$ and $\leq_{T_2}$ are contained within $\leq_{\mathcal{T}_{eq}}$ and are thereby non-contradicting. □

6.2 λXP Queries

We present λXP a hybrid logic that combines navigation with standard programming language constructs.

6.2.1 Philosophy of Sets and Functions

Sets and functions can be seen as relations which may contain infinitely many elements. We are not only interested in finite sets, since we may also want to define sets of strings and not only sets of nodes.

Therefore, one must clarify what it means that an algorithm returns a set or a function. A set will be given by an algorithm that enumerates the elements of the set. This algorithm can then be used to evaluate membership expression $x \in E$, where all possible bindings of x to some value of E must be generated. A function f of type $T \to T'$ will be given by an algorithm that inputs an argument v of the function and returns the value of $f(v)$.

It should be noticed that sets are considered as generators, while functions are considered as applications. Therefore, it will not be possible to encode sets by their characteristic functions in the well-moded subset of our language, to which the evaluator will be restricted.

6.2.2 Constants

The language λXP is parameterized by a set of typed constants $\mathcal{C}onst$ that are listed in Figure 6.2. The type of $c \in \mathcal{C}onst$ is denoted by $\textsc{Type}(c)$. The semantics of a constant c of type T with respect to a sequence of trees s with node π is value $[\![c]\!]_{s,\pi} \in [\![T]\!]_s$ that is defined in Figure 6.3.

In order to convert a list into a set or vise versa, we need to impose a total order on the elements, in which the conversion is carried out. The order is most relevant for converting a finite set into a list in a deterministic manner. Which precise order to chose is irrelevant in such cases. And whenever a concrete order is expected, it can be obtained by applying a sort function to the list obtained from the set. The ordering chosen for the conversions may be implementation dependent. This happens for instance, when converting a set to a list, where the elements of the set are produced in streaming manner. One then wants to apply the conversion to the elements of the set in the order in which they arrive on the stream. Sorting these elements in some other order would require to buffer the whole set, which may block real time progress in a streaming setting or require high memory consumption.

The current node which changes during evaluation of various axes can always be accessed by applying function *cur* to the unit, i.e. to the empty tuple () which has type **none**: the call will return the node π of the current environment.

Furthermore, we give in the following one of many possible implementations of constant *listconcat* that concatenates lists of type T to obtain a single list of type T: This implementation models lists, as open lists L^{op} of type T that are lists L with a future element $\bullet$, for example, $v_1 :: \ldots :: v_n :: \bullet$, where $v_i \in [\![T]\!]_s$. Then *listconcat* can be implemented as follows.

$$\mathit{listconcat}(L) =_{\text{df}} \textbf{ let } O = \mathit{open\text{-}list.new}() \textbf{ in } \mathit{open\text{-}list.toList}(\mathit{listconcat}'(L, O))$$

where

$$\mathit{open\text{-}list.new} =_{\text{df}} \textbf{let } F = \mathit{future.new}() \textbf{ in } (F, F) \ ,$$

$$\mathit{open\text{-}list.toList}(O) =_{\text{df}} \textbf{let } (OL, F) = O, b = \mathit{future.bind}(F, \mathit{nil}) \textbf{ in } OL \ ,$$

$$\begin{aligned} \mathit{listconcat}'(L, O) = \ & \mathit{case}\ L\ \mathit{of}\ \mathit{nil}\ \mathit{then}\ \mathit{return}\ O \\ & \mathit{case}\ L\ \mathit{of}\ H : L'\ \mathit{then}\ \mathit{return} \\ & \quad \mathit{listconcat}'(L', \mathit{open\text{-}list.extend}(O, H)) \end{aligned} \ ,$$

Constants	Type
$1, 2, 3, \ldots, max\_int$	$\mathbf{int} \preceq \mathbf{num}$
$\text{-}1E4, 1.23, \text{-}0, 0, \ldots, max\_float$	$\mathbf{float} \preceq \mathbf{num}$
$c \in \Delta$	$\mathbf{char}$
$w \in \Delta^*$	$\mathbf{string}$
$true, false$	$\mathbf{bool}$
$+, -, \times, \div, idiv, mod$	$\bigcup_{T \preceq \mathbf{num}} T \times T \to T$
$\neq_T, =_T, >_T, <_T, \geq_T, \leq_T$ T is equality type	$T \times T \to \mathbf{bool}$
nil	$[\mathbf{none}]$
$\emptyset$	$\{\mathbf{none}\}$
cur	$\mathbf{unit} \to \mathbf{node}$
$set2list_T$	$\{T\} \to [T]$
$list2set_T$	$[T] \to \{T\}$
$\in_T$	$T \times \{T\} \to \mathbf{bool}$
$labtype$	$\mathbf{node} \to \mathbf{lab}$
$labvalues$	$\mathbf{node} \to \cup_{\sigma^{(k)} \in \Sigma} \mathbf{string}^k$
$type_{T,T'}$	$T \times T' \to \mathbf{bool}$
$cast_{T,T'}$	$T \to T'$
$root$	$\mathbf{string} \to \mathbf{node}$
$roots$	$\mathbf{string} \to [\mathbf{node}]$
$listconcat$	$[[T]] \to [T]$

Figure 6.2: Constants.

$$
\begin{aligned}
\mathit{open\text{-}list.extend}(O, H) =_{\text{df}} \; & \mathit{case}\ H\ \mathit{of}\ \mathit{nil}\ \mathit{then}\ \mathit{return}\ O \\
& \mathit{case}\ H\ \mathit{of}\ X :: H'\ \mathit{then} \\
& \quad \mathit{let}\ F' = \mathit{future.new}(), \\
& \qquad (OL, F) = O, \\
& \qquad b = \mathit{future.bind}(F, X :: F') \\
& \quad \mathbf{in}\ (OL, F')
\end{aligned}
$$

6.2.3 Syntax

The abstract syntax of λXP formulas is given in Figure 6.4. It is parameterized by some alphabet Σ, and alphabet Δ, the set of constants $\mathcal{C}onst$, a set of variables $\mathcal{V}$, a set of axes A, and a set of label properties $\mathcal{L}$. A label property $L \in \mathcal{L}$ verifies a property for some label in Σ, i.e. $L : \Sigma \to \mathbb{B}$. For some sequence of data trees s the label properties $L \in \mathcal{L}$ are used in unary relation symbols and interpreted as the set L^s of nodes of s whose label satisfy L.

The *navigational* fragment F of λXP are formulas composed of axes, label properties, node tests, conjunction, disjunction, and negation only:

$$F ::= A(F) \mid L \mid is_x \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F$$

$[\![i]\!]_{s,\pi} = i$	if $\textsc{Type}(i) = \mathbf{num} \cup \mathbf{char} \cup \mathbf{string} \cup \mathbf{bool}$
$[\![o]\!]_{s,\pi} = \bigcup_{T \preceq \mathbf{num}} (o^{[\![T]\!]_s})^\perp$	if $\textsc{Type}(o) = \bigcup_{T \preceq \mathbf{num}} T \times T \to T$
$[\![o_T]\!]_{s,\pi} = (o^{[\![T]\!]_s})^\perp$	if $\textsc{Type}(o) = T \times T \to \mathbf{bool}$
$[\![nil]\!]_{s,\pi} =$ empty list	
$[\![\emptyset]\!]_{s,\pi} =$ empty set	
$[\![set2list]\!]_{s,\pi} = (set2list^{[\![T]\!]_s})^\perp$	set to list conversion
$[\![list2set]\!]_{s,\pi} = (list2set^{[\![T]\!]_s})^\perp$	list to set conversion
$[\![cur]\!]_{s,\pi} = \{(\_, \pi)\}$	always returns the current node
$[\![\in_T]\!]_{s,\pi} = (\in^{[\![T]\!]_s})^\perp$	set membership
$[\![labtype]\!]_{s,\pi} = (labtype)^\perp$	node label types
$[\![labvalues]\!]_{s,\pi} = (labvalues)^\perp$	node label data values
$[\![type_{T,T'}]\!]_{s,\pi} = (type^{[\![T]\!]_s,[\![T']\!]_s})^\perp$	checks whether for $v : T$ also $v : T'$
$[\![cast_{T,T'}]\!]_{s,\pi} = (cast^{[\![T]\!]_s,[\![T']\!]_s})^\perp$	casts from T to T'
$[\![root]\!]_{s,\pi} = (root)^\perp$	root nodes by strings
$[\![roots]\!]_{s,\pi} = (roots)^\perp$	list of root nodes by strings
$[\![listconcat]\!]_{s,\pi} = (listconcat^{[\![T]\!]_s})^\perp$	concatenation of lists

Figure 6.3: Semantic of constants from Figure 6.2.

When furthermore only forward axes child, descendant, and following-sibling are used, i.e. $A = \{ch, ch^+, ns^+\}$, then formulas belong to the forward navigational fragment of λXP, called Fxp.

Besides conjunction, disjunction, and negation λXP contains also constants c, variables x, existential quantification, and goto constructs $\mathbf{at}(E_1, E_2)$. Standard programming language constructs that λXP inherits are conditionals and the creation and decomposition of tuples, sets, and lists. λXP can define *recursive higher-order* functions, for which it contains a recursive let-construct. Notice that all expressions bound to variables must be monotone, see next section, since otherwise we can not ensure the termination of recursively defined functions. Finally λXP contains expressions for function applications and definitions. λXP contains try and catch expression for exception handling, which we do not consider in our formal analysis.

6.2.4 Positive Contexts

In order to decide whether an expression with recursive let definitions is valid or not, we introduce positive contexts of expressions and positive functions: We define *positive contexts* and *positive functions* in Figure 6.5.

6.2.5 Type System

The typing rules of λXP are explained in Figure 6.6. The type of constants c and variables x is denoted by $\textsc{Type}(c)$ and $\textsc{Type}(x)$, respectively. Most of the typing

E	::=	$E_1 \wedge E_2 \mid E_1 \vee E_2 \mid \neg E \mid c \mid x \mid \exists x.E$	first-order logic
	$\mid$	$A(E) \mid L \mid \mathbf{at}(E_1, E_2)$	navigation
	$\mid$	$\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3$	conditionals
	$\mid$	$\mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ E$	recursion
		where all x_i occur in positive contexts in $E_1, \ldots, E_n$	
		and all E_i have function types	
	$\mid$	$(E_1, \ldots, E_n) \mid \mathbf{match}\ E_1\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ E_2$	tuples where $n \neq 1$
	$\mid$	$\{x \mid E\}$	sets
	$\mid$	$E_1 :: E_2 \mid \mathbf{match}\ E_1\ \mathbf{with}\ x :: y\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3$	lists
	$\mid$	$E_1 E_2 \mid \lambda x.\ E$	functions
	$\mid$	$\mathbf{try}\ E_1\ \mathbf{catch}(x)\ E_2 \mid \mathbf{raise}\ E$	exceptions

Figure 6.4: Abstract syntax of λXP formula E, where $c \in \mathcal{C}onst$ is a constant, $x \in \mathcal{V}$ a variable, $L \in \mathcal{L}$ a label predicate, and A a set of axes.

rules are standard. Notice that we require explicit type annotations on function definition, since otherwise this type cannot always be inferred. Notice also that for recursive let definitions we require that all variables are function types.

6.2.6 Library Functions

A common shortcut for the test whether the value of a variable is equal to the current node is

$$is_x =_{\mathrm{df}} x = cur().$$

Another shortcut is the simple non-recursive let that is reduced to function application:

$$\mathbf{let}\ x = E\ \mathbf{in}\ E' =_{\mathrm{df}} (\lambda x.E')(E)$$

In Figure 6.7, we define sets of tuples and functions that input tuples. Note also, that one can always add or remove parenthesis in (E), so this expression is equivalent to E; there are no 1-tuples in the language.

In Figure 6.8, we define the usual notion for sets, including enumerations for finite sets, unions, and maps, and for-loops for their aggregation. Finally, we can convert sets into lists, based on an implementation dependent order on the elements of the set.

Data joins are defined in Figure 6.9. They are reduced to existential quantifiers, conjunctions, and data comparisons. The usual list functions map, foldL, and append are defined in Figure 6.10.

$$
\begin{array}{rcl}
C & ::= & [.] \mid C \wedge E \mid E \wedge C \mid C \vee E \mid E \vee C \\
& \mid & \in_T (E, C) \mid \exists x.C \mid A(C) \mid \mathbf{at}(E, C) \mid \mathbf{at}(C, E) \\
& \mid & \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ E' \mid \mathbf{if}\ E\ \mathbf{then}\ E'\ \mathbf{else}\ C \\
& \mid & \mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ C \\
& & \quad \text{where all } x_i \text{ occur in positive contexts in } E_1, \ldots, E_n \\
& & \quad \text{and all } E_i \text{ have function types} \\
& \mid & \mathbf{match}\ C\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ E \\
& & \quad \text{where all } x_i \text{ occur in positive contexts in } E \\
& \mid & \mathbf{match}\ E\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ C \\
& \mid & \{x \mid C\} \\
& \mid & C :: E \mid E :: C \\
& \mid & \mathbf{match}\ C\ \mathbf{with}\ x :: y\ \mathbf{then}\ E\ \mathbf{else}\ E' \\
& & \quad \text{where } x \text{ and } y \text{ occur in positive contexts in } E \\
& \mid & \mathbf{match}\ E\ \mathbf{with}\ x :: y\ \mathbf{then}\ C\ \mathbf{else}\ E' \\
& \mid & \mathbf{match}\ E\ \mathbf{with}\ x :: y\ \mathbf{then}\ E'\ \mathbf{else}\ C \\
& \mid & PC \qquad \text{where } P \text{ is a positive function} \\
P & ::= & \lambda x.E \qquad \text{where } x \text{ occurs only only in positive contexts of } E \\
& \mid & c \qquad \text{where } [\![c]\!]_s \text{ is Scott continuous for all } s \text{ such as } +, *, \ldots \\
& \mid & \mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ x_j \\
& & \quad \text{where all } x_i \text{ occur in positive contexts in } E_1, \ldots, E_n \\
& & \quad \text{and } E_j \text{ is some positive function } P' \text{ and } 1 \leq j \leq n
\end{array}
$$

Figure 6.5: Positive contexts C and positive function P in λXP formulas.

$$\frac{E_i : \mathbf{bool}}{E_1 \wedge E_2 : \mathbf{bool}} \qquad \frac{E_i : \mathbf{bool}}{E_1 \vee E_2 : \mathbf{bool}} \qquad \frac{E : \mathbf{bool}}{\neg E : \mathbf{bool}} \qquad \frac{\textsc{Type}(c) = T}{c : T}$$

$$\frac{\textsc{Type}(x) = T}{x : T} \qquad \frac{x : T \quad E : \mathbf{bool}}{\exists x.E : \mathbf{bool}}$$

$$\frac{E : \mathbf{bool}}{A(E) : \mathbf{bool}} \qquad \frac{\mathit{true}}{L : \mathbf{bool}} \qquad \frac{E_1 : \mathbf{node} \quad E_2 : T}{\mathbf{at}(E_1, E_2) : T}$$

$$\frac{E_1 : \mathbf{bool} \quad E_2 : T \quad E_3 : T}{\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : T}$$

$$\frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad E_1 : T_1 \quad \ldots \quad E_n : T_n \quad E : T \quad T_1, \ldots, T_n \text{ function types}}{\mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ E : T}$$

$$\frac{E_1 : T_1 \quad \ldots \quad E_n : T_n}{(E_1, \ldots, E_n) : T_1 \times \ldots \times T_n}$$

$$\frac{E_1 : T_1 \quad \ldots \quad E_n : T_n \quad E_1 : T_1 \times \ldots \times T_n \quad E_2 : T}{\mathbf{match}\ E_1\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ E_2 : T}$$

$$\frac{x : T \quad T \in \mathcal{T}_{eq} \quad E : \mathbf{bool}}{\{x \mid E\} : \{T\}}$$

$$\frac{E_1 : T_1 \quad E_2 : [T_2]}{E_1 :: E_2 : [T_1 \cup T_2]} \qquad \frac{E_1 : [T] \quad E_2, E_3 : T'}{\mathbf{match}\ E_1\ \mathbf{with}\ x :: y\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : T'}$$

$$\frac{E_1 : T_1 \to T_2 \quad E_2 : T_1}{E_1 E_2 : T_2} \qquad \frac{x : T_1 \quad E : T_2 \quad \lambda x : T_1 \to T_2}{\lambda x.E : T_1 \to T_2}$$

$$\frac{E : T \quad T \preceq T' \quad T' \text{ admissible}}{E : T'}$$

Figure 6.6: Typing rules.

6.2.7 Recursive Functions

We can define recursive functions such as the faculty function as done usually in functional programming:

$$\mathit{fac} =_{\mathrm{df}} \mathbf{letrec}\ f = \lambda y.\ \mathbf{if}\ y = 0\ \mathbf{then}\ 1\ \mathbf{else}\ y \cdot f(y-1)\ \mathbf{in}\ f$$

This definition relies on the laziness of conditionals. For instance, when computing $f(0)$ then in the second branch of the conditional there is the expression $f(-1)$, which is undefined and evaluates to $\bot$. But this second branch will not be evaluated since the test of the conditional $y = 0$ is satisfied in this case.

6.2.8 Regular Axes

We would like to notice that within λXP we can express regular axis E^* where $E : \{\mathbf{node}\}$, which recursively navigates from the current node to one of the nodes selected by E:

$$\begin{aligned} E^* =_{\mathrm{df}} \mathbf{letrec}\ O = \lambda X.\ \mathbf{let}\ X' = X \cup \{y \mid \exists x.x \in X \wedge y \in \mathbf{at}(x, E)\} \\ \mathbf{in\ if}\ X = X'\ \mathbf{then}\ X'\ \mathbf{else}\ O(X') \\ \mathbf{in}\ O(\{\mathit{cur}()\}) \end{aligned}$$

$(E) =_{\text{df}} E$
$\{(x_1, \ldots, x_n) \mid E\} =_{\text{df}} \{x \mid \exists x_1 \ldots \exists x_n.\ x = (x_1 \ldots, x_n) \wedge E\}$
$\lambda(x_1, \ldots, x_n).\ E =_{\text{df}} \lambda z.\mathbf{match}\ z\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ E$

Figure 6.7: Tuples where $n \neq 1$.

$\{e\} =_{\text{df}} \{x \mid x = e\}$
$E_1 \cup E_2 =_{\text{df}} \{x \mid x \in E_1 \vee x \in E_2\}$
$E_1 \cap E_2 =_{\text{df}} \{x \mid x \in E_1 \wedge x \in E_2\}$
$E_1 \setminus E_2 =_{\text{df}} \{x \mid x \in E_1 \wedge \neg(x \in E_2)\}$
$\mathit{map\text{-}set} =_{\text{df}} \lambda f \lambda x.\ \mathit{list2set}(\mathit{map}(f, \mathit{set2list}(x)))$
$\mathit{foldL\text{-}set} =_{\text{df}} \lambda f.\lambda x.\ \lambda y.\mathit{list2set}(\mathit{foldL}(\mathit{set2list}(y), x, f))$

Figure 6.8: Sets.

Notice, that the recursive definition of regular axes and the one for datalog as we show below, has a terminating condition. When one of these computed sets during some recursion does not grow anymore, meaning that it is the same with respect to the previous recursion, then one stops and the final set is computed. Notice also that a similar definition for recursion functions, as in the previous section, does not work. One could imagine to compute the graph of all arguments and values of the function, but since it may not be finite, one cannot test for equality.

6.2.9 Datalog

We can express save Datalog programs by considering predicates as sets of tuples. For instance consider the following datalog program defining the accessibility relation of a graph with edge relation e.

$$p(x, z) :- e(x, z).$$
$$p(x, z) :- p(x, y), p(y, z).$$

With recursive functions, such recursive set definitions can be expressed as follows:

$$\begin{array}{l}\mathbf{letrec}\ f = \lambda p'.\ \mathbf{let}\ p = p' \cup \{(x, z) \mid e(x, z) \vee \exists y.\ (x, y) \in p' \wedge (y, z) \in p'\} \\ \qquad\qquad\qquad \mathbf{in\ if}\ p = p'\ \mathbf{then}\ p\ \mathbf{else}\ f(p) \\ \mathbf{in}\ f(\emptyset)\end{array}$$

$$\exists^{\bowtie}(E_1, E_2) =_{\text{df}} \exists y_1.\exists y_2.y_1 \in E_1 \wedge y_2 \in E_2 \wedge y_1 \bowtie y_2$$

Figure 6.9: Data comparison where $\bowtie \in \{\neq, =, >, <, \geq, \leq\}$.

$$\begin{aligned}
\mathit{map} &=_{\text{df}} \textbf{letrec } \mathit{map}' = \lambda f \lambda x.\ \textbf{match } x \textbf{ with } z :: x' \textbf{ then} \\
&\qquad f(z) :: \mathit{map}'\ f\ x' \textbf{ else } \mathit{nil} \textbf{ in } \mathit{map}' \\
\mathit{foldL} &=_{\text{df}} \textbf{letrec } \mathit{foldL}' = \lambda f.\lambda x.\lambda y.\ \textbf{match } y \textbf{ with } z :: y' \textbf{ then} \\
&\qquad \mathit{foldL}'\ f\ f(x, z)\ y' \textbf{ else } x \textbf{ in } \mathit{foldL}' \\
\mathit{foldR} &=_{\text{df}} \textbf{letrec } \mathit{foldR}' = \lambda f.\lambda x.\lambda y.\ \textbf{match } y \textbf{ with } z :: y' \textbf{ then} \\
&\qquad f(z, \mathit{foldR}' f\ x\ y') \textbf{ else } x \textbf{ in } \mathit{foldR}' \\
\mathit{append} &=_{\text{df}} \textbf{letrec } \mathit{append}' = \lambda(x, y).\textbf{match } x \textbf{ with } z :: x' \textbf{ then} \\
&\qquad z :: \mathit{append}'(x', y) \textbf{ else } y \textbf{ in } \mathit{append}'
\end{aligned}$$

Figure 6.10: Lists.

6.2.10 Data Comparisons

Consider the following example that makes references to two different input trees in $s = t_1, t_2$, where $x_1 = \mathit{root}(t_1)$ and $x_2 = \mathit{root}(t_2)$:

$$\begin{aligned}
(Q_1)\quad & \textbf{at}(x_1, \{x \mid \mathit{ch}^+(\texttt{tweet} \wedge \mathit{is}_x \\
& \quad \wedge \exists^{=}(\texttt{fn:data}(\mathit{set2list}(\{x' \mid \mathit{ch}(\texttt{user_name} \wedge \mathit{is}_{x'})\})), \\
& \qquad \textbf{at}(x_2, \texttt{fn:data}(\\
& \qquad\quad \mathit{set2list}(\{x' \mid \mathit{ch}^+(\texttt{students} \wedge \mathit{ch}(\texttt{name} \wedge \mathit{is}_{x'}))\}))))))\})
\end{aligned}$$

It selects the set of `tweet`-descendant nodes in t_1, that have a `user_name`-child whose typed value is equal to the typed value of `name`-children of `student`-descendants of t_2. It is a join query between t_1 and t_2, where **at**-formulas are used to jump between the two trees. It corresponds to XPath query `$`x_1`//tweet[user_name = $`x_2`//students/name]`. The typed values are obtained via atomization that we discussed in Section 5.2.4, while we refer to Section 7.2.4 for an λXP expression defining `fn:data`.

6.3 Semantics

We next define a denotational semantics for λXP queries. Since λXP permits recursive definitions, we will base the semantics least fixed points, whose existence is ensured by Kleene's fixed point theorem.

6.3.1 Complete Partial Orders

For any s, we will turn the set of all s-values into a CPO, so that Kleene's fixed point theorem can be applied.

$$\frac{}{\bot \sqsubset v} \qquad \frac{}{\mathit{false} \sqsubset \mathit{true}}$$

$$\frac{v, v' \in [\![T]\!]_s \quad T \text{ set type}}{v \subsetneq v' \Rightarrow v \sqsubset v'} \qquad \frac{v, v' \in [\![T]\!]_s \quad T \text{ function type}}{v \subsetneq v' \Rightarrow v^{\bot} \sqsubset v'^{\bot}}$$

Figure 6.11: Partial order $\sqsubset$ on s-values, valid for all $v, v' \in V_s \setminus \{\bot\}$.

The main difficulty that we have to deal with here is that of program errors or nontermination in our denotational semantics. As usual, we will represent these by the special symbol $\bot$, and add $\bot$ to the s-values in an appropriate manner.

For all types T except for function types, we define the domain of the CPO by adding $\bot$ to the s-values of type T:

$$[\![T]\!]_s^{\bot} = [\![T]\!]_s \cup \{\bot\}$$

For function types $T \to T'$, we will define the CPOs by lifting functions to undefined inputs: A partial function $f : [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s \to [\![T]\!]_s^{\bot}$ is lifted to a total function $f^{\bot} : [\![T_1]\!]_s^{\bot} \times \ldots \times [\![T_n]\!]_s^{\bot} \to [\![T]\!]_s^{\bot}$ such that for all $v_1 \in [\![T_1]\!]_s^{\bot}, \ldots, v_n \in [\![T_n]\!]_s^{\bot}$:

$$f^{\bot}(v_1, \ldots, v_n) = \begin{cases} f(v_1, \ldots, v_n) & \text{if } v_1, \ldots, v_n \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

We define the following domains for function types:

$$[\![T \to T']\!]_s^{\bot} = \{f^{\bot} \mid f \in [\![T \to T']\!]_s\} \cup \{\bot\}$$

Notice that we distinguish the empty partial functions $f_{\emptyset} : T \to T'$ from $\bot$.

For any sequence s we now define a partial order $\sqsubset$ on the set of all lifted s-values $V_s = \cup_{T \text{ type}} [\![T]\!]_s^{\bot}$. This is the least partial order that satisfies the axioms in Figure 6.11.

According to the first rule, $\bot$ is the least element of V_s with respect to $\sqsubset$. For Booleans we assume that $\mathit{false} \sqsubset \mathit{true}$. Sets do increase if an element is added, and similar for functions. Tuples increase if one of their components increases.

Since $\mathit{false} \sqsubset \mathit{true}$ it follows that $\wedge^{\mathbf{bool}}$ and $\vee^{\mathbf{bool}}$ are monotone functions while $\neg^{\mathbf{bool}}$ is not monotone. Since sets increase when adding elements, it follows that set union is a monotone function, while set complement is not. The membership functions $\Lambda v'.v \in v'$ is monotone for any v, while non-membership $\Lambda v'.v \notin v'$ is not.

Proposition 2. *For any function $f : [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s \to [\![T]\!]_s^{\bot}$ and any $1 \leq j \leq n$, if f is monotone in its j-th argument then the function $f^{\bot}$ is monotone in its j'th argument too..*

Proof. We consider two members of $[\![T_1]\!]_s^{\bot} \times \ldots \times [\![T_n]\!]_s^{\bot}$, $v = (v_1, \ldots, v_n)$ and $v' = (v_1, \ldots, v_{j-1}, v'_j, v_{j+1}, \ldots, v_n)$ such that $v_j \sqsubset v'_j$. We have to show that $f^{\bot}(v) \sqsubseteq f^{\bot}(v')$. If there exists $1 \leq i \leq n$ such that $v_i = \bot$, then $f^{\bot}(v) = \bot \sqsubseteq f^{\bot}(v')$.

Otherwise, $v_i \neq \bot$ for all $1 \leq i \leq n$ and also $v'_j \neq \bot$. Hence for all $1 \leq i \leq n$, $v_i \sqsubset v'_i$ in the CPO $[\![T_i]\!]_s$ ordered by the restriction of $\sqsubset$. The monotonicity of f then yields $f^{\bot}(v) = f(v) \sqsubseteq f(v') = f^{\bot}(v')$. □

We will show next that the ordering $\sqsubset$ preserves the "top-level structure" of s-values different from the error: it increases only some leaves containing functions or Booleans, or adds some elements to the sets.

Definition 8. *An* ordered top-level structure *is a term with n-constructors from*

$$\{set^{(n')} \mid n' \in \mathbb{N}_0 \cup \{\infty\}\} \cup \{tuple^{(n)}, list^{(n)} \mid n \in \mathbb{N}_0\}$$

and constants from $\cup_{T \in \mathcal{T}_{atom}} [\![T]\!]_s \cup \cup_{T,T' \text{ types}} [\![T \to T']\!]_s$.

To any value of $V_s \setminus \{\bot\}$ we can assign a set of ordered top-level structures, by fixing for each set an order on the elements.

Lemma 6. *For any s and any values $v, v' \in V_s$, if $v \sqsubseteq v'$ then either $v = \bot$ or there exist ordered top-level structures t of v and t' of v' such that for all nodes π of t and $n \geq 0$:*

1. *if $t[\pi] = set^{(n)}$ then $t'[\pi] = set^{(m)}$ for some $n \leq m$,*
2. *if $t[\pi] = t'[\pi]$,*
3. *if $t[\pi] = false$ and $t'[\pi] = true$,*
4. *if $t[\pi] = \tilde{v}^{\bot}$ and $t[\pi'] = \tilde{v}'^{\bot}$ are functions such that $\tilde{v} \subseteq \tilde{v}'$.*

Proof. By structural induction on derivations of $v \sqsubseteq v'$. □

Proposition 3. *For any s, the partial order $\sqsubset$ on s-values is a complete partial order (CPO).*

Proof. We need to show that any increasing nonempty chain $v_1 \sqsubseteq \ldots v_i \sqsubseteq \ldots$ has a least upper bound. The chain may be of finite or infinite length.

Only the elements in the prefix of a prefix of the chain may be equal to $\bot$, since only the first rule concerns $\bot$. In this case, the least upper bound of the same chain but without the $\bot$-prefix is a least upper bound for the complete chain too. Therefore, we can assume that $\bot$ does not occur in the chain. The only chains containing an atom must be constantly equal to this atom, so it is the least upper bound. Therefore, it is sufficient to consider chains without values of atomic types or $\bot$.

Now suppose that all v_i are either tuples, sets, lists, or functions. According to Lemma 6, there exist ordered top-level structures t_i of v_i, such that the nodes sets of t_i are increasing as well as the labels of the same nodes.

Let t be the ordered top-level structure with $\text{nodes}(t) = \cup_i \text{nodes}(t_i)$, such that for every node π of some t_i, the label of t is $t[\pi] = \sqcup_{j \geq i} t_i[\pi]$. In the case where $t_i[\pi] = set^{(n_i)}$ for some n_i, we set $\sqcup_{j \geq i} t_i[\pi] = \sqcup_{j \geq i} set^{(n_i)} = set^{\sup_{j \geq i} n_j}$. Note that

$\sup_{j \geq i} n_j$ may be equal to ∞. The only other possibility applies to leaf nodes π, where all $t_i[\pi]$ are some functions $f_i^\perp$ where $f_i \in [\![T]\!]_s$. In this case, $\sqcup_i t_i[\pi] = (\cup_i f_i)^\perp$. The unique value v that admits t as ordered top-level structure is the least upper bound of the chain of v_i's. □

It follows from Proposition 3 that the partial order $\sqsubset$ restricted to $V_s \setminus \{\perp\}$ forms a CPO as well.

Proposition 4. *For any function $f : [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s \to [\![T]\!]_s^\perp$ that is Scott-continuous in all arguments, the function $f^\perp$ is also Scott-continuous in all arguments.*

Proof. We fix a sequence of values $v_1, \ldots, v_n \in V_s$ and $1 \leq i \leq n$. We define a function g by:

$$g(v) = f(v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n)$$

for all $v \in V_s$. By assumption, g is Scott-continuous, i.e., for any increasing nonempty chain $v'_1 \sqsubseteq \ldots \sqsubseteq v'_i \sqsubseteq \ldots$ in $V_s \setminus \{\perp\}$ it holds that $\sqcup_i g(v'_i) = g(\sqcup_i v'_i)$. We have to show that $g^\perp$ is Scott-continuous too. So let $v'_1 \sqsubseteq \ldots \sqsubseteq v'_i \sqsubseteq \ldots$ be an increasing chain in V_s. If $v'_i = \perp$ for all i we have $\sqcup_i g^\perp(v'_i) = \sqcup_i g^\perp(\perp) = \sqcup_i \perp = \perp = g^\perp(\sqcup_i v'_i)$.

Otherwise, we can remove the maximal prefix of $\perp$-values from the chain, and obtain an increasing nonempty chain in $V_s \setminus \{\perp\}$. Let m be the first index of this chain. We then have $\sqcup_{i \geq m} g(v'_i) = g(\sqcup_{i \geq m} v'_i)$ since g is Scott-continuous. Furthermore, $\sqcup_i g^\perp(v'_i) = \sqcup_{i \geq m} g^\perp(v'_i)$ and $\sqcup_i v'_i = \sqcup_{i \geq m} v'_i$. Hence $\sqcup_i g^\perp(v'_i) = g^\perp(\sqcup_i v'_i)$ as required. □

6.3.2 Denotational Semantics

In Figure 6.12 and Figure 6.13 we define the denotational semantics $[\![E]\!]_{s,\pi,\mu}$ of an λXP formula E with respect to a sequence of data trees s, a node $\pi \in \text{nodes}(s)$ of one of these data trees, and a possibly partial variable assignment μ from variables to s-values. We define the consistent union of two variable assignments μ and μ' by $\mu \uplus \mu' = \mu \cup \mu'$ if this union is a functional relation and leave it undefined otherwise. We also write $\mu \uplus x/\perp$ instead of μ if x is undefined in μ. This is a slight abuse of notation, in that μ is a functional relation with $\mu \subseteq V \times V_s \setminus \{\perp\}$ rather than a total function with codomain V_s.

Furthermore, during the computation of a set $[\![E]\!]_{s,\pi,\mu}$ where $E : \{T\}$, the order in which the elements of the set are discovered will depend on s, π and μ. In order to obtain a deterministic semantics, we will fix the required orders as parameters of the semantics. As parameters of the denotational semantics, we assume for any equality type $T \in \mathcal{T}_{eq}$, expression $E : \{T\}$, trees s with node π, and variable assignment μ into s a total order $\leq_{E,T,s,\pi,\mu}$ on the elements of $[\![E]\!]_{s,\pi,\mu}$. This order can be freely chosen by the implementation.

$$
\begin{array}{l}
[\![E_1 \wedge E_2]\!]_{s,\pi,\mu} = [\![E_1]\!]_{s,\pi,\mu} \wedge^{Bool_\perp} [\![E_2]\!]_{s,\pi,\mu} \\
[\![E_1 \vee E_2]\!]_{s,\pi,\mu} = [\![E_1]\!]_{s,\pi,\mu} \vee^{Bool_\perp} [\![E_2]\!]_{s,\pi,\mu} \\
[\![\neg E]\!]_{s,\pi,\mu} = \neg^{Bool_\perp} [\![E]\!]_{s,\pi,\mu} \\
[\![c]\!]_{s,\pi,\mu} = [\![c]\!]_{s,\pi} \\
[\![x]\!]_{s,\pi,\mu} = \mu(x) \qquad \text{may be undefined if } x \notin dom(\mu) \\
[\![\exists x.E]\!]_{s,\pi,\mu} = \begin{cases} \mathit{true} & \text{if exists } v \in [\![\textsc{Type}(x)]\!] \text{ such that } [\![E]\!]_{s,\pi,\mu \uplus [x/v]} = \mathit{true} \\ \mathit{false} & \text{if all } v \in [\![\textsc{Type}(x)]\!] \text{ satisfy } [\![E]\!]_{s,\pi,\mu \uplus [x/v]} = \mathit{false} \\ \perp & \text{otherwise} \end{cases} \\
[\![A(E)]\!]_{s,\pi,\mu} = (\exists \pi' \in \text{nodes}(s).\ A^s(\pi,\pi') \wedge^{Bool_\perp} [\![E]\!]_{s,\pi',\mu} = \mathit{true}) \\
[\![L]\!]_{s,\pi,\mu} = (\pi \in L^s) \\
[\![\mathbf{at}(E_1,E_2)]\!]_{s,\pi,\mu} = (\Lambda \pi \in \text{nodes}(s).\ [\![E_2]\!]_{s,\pi,\mu})^\perp([\![E_1]\!]_{s,\pi,\mu}) \\
[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3]\!]_{s,\pi,\mu} = \begin{cases} [\![E_2]\!]_{s,\pi,\mu} & \text{if } [\![E_1]\!]_{s,\pi,\mu} = \mathit{true} \\ [\![E_3]\!]_{s,\pi,\mu} & \text{if } [\![E_1]\!]_{s,\pi,\mu} = \mathit{false} \\ \perp & \text{if } [\![E_1]\!]_{s,\pi,\mu} = \perp \end{cases} \\
[\![\mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ E]\!]_{s,\pi,\mu} = \\
\quad \mathbf{Letrec}\ v_1 = [\![E_1]\!]_{s,\pi,\mu'}, \ldots, v_n = [\![E_n]\!]_{s,\pi,\mu'}\ \mathbf{in}\ [\![E]\!]_{s,\pi,\mu'} \\
\quad \text{where } \mu' = \mu \uplus [x_1/v_1, \ldots, x_n/v_n]
\end{array}
$$

Figure 6.12: Part 1 of denotational semantics of λXP, where s is a sequence of data trees, $\pi \in \text{nodes}(s)$, and μ to s-values is a well-typed variable assignment.

6.3.3 Well-definedness

For instance, one can verify with this definition that the mutual recursive definition:

$$\mathbf{letrec}\ g = \lambda z.f(z), f = \lambda y.27\ \mathbf{in}\ g(0)$$

evaluates to 27. The fixed point operator for the two equations will start with $(\emptyset,\emptyset)$, then produce $(\emptyset, \Lambda v.27)$ and finally converge to $(\Lambda v.27, \Lambda v.27)$. Note that the semantics must use tuples with increasing components to obtain the correct evaluation.

The following proposition states that the semantics of **letrec** expressions is always well-defined.

Proposition 5. *Let $x_1 : T_1, \ldots, x_n : T_n$ be variables of function types, s be a sequence of trees, $\pi \in nodes(s)$ and μ a variable assignment to s-values. Let $D = [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s$ and $\sqsubseteq_D$ be the least partial order on D satisfying for any two elements $(v_1, \ldots, v_n) \in D$ and $(v'_1, \ldots, v'_n) \in D$:*

$$\frac{v_1 \sqsubseteq v'_1 \quad \ldots \quad v_n \sqsubseteq v'_n}{(v_1, \ldots, v_n) \sqsubseteq_D (v'_1, \ldots, v'_n)}$$

Let $\sqsubset_D$ be the irreflexive restriction of $\sqsubseteq_D$. Then $(D, \sqsubset_D)$ is a CPO *with least element $(\emptyset, \ldots, \emptyset)$.*

$$
\begin{array}{l}
[\![(E_1,\ldots,E_n)]\!]_{s,\pi,\mu} = \qquad\qquad \text{where } E_1 : T_1,\ \ldots,\ E_n : T_n \\
\ (\Lambda v_1 \in [\![T_1]\!]_s, \ldots, v_n \in [\![T_n]\!]_s.\ (v_1,\ldots,v_n))^{\perp}([\![E_1]\!]_{s,\pi,\mu}, \ldots, [\![E_n]\!]_{s,\pi,\mu}) \\
[\![\textbf{match } E_1 \textbf{ with } (x_1,\ldots,x_n) \textbf{ in } E_2]\!]_{s,\pi,\mu} = \\
\left\{ \begin{array}{ll} [\![E_2]\!]_{s,\pi,\mu \uplus [x_1/v_1,\ldots,x_n/v_n]} & \text{if } [\![E_1]\!]_{s,\pi,\mu} = (v_1,\ldots,v_n) \\ \perp & \text{otherwise} \end{array} \right. \\
[\![\{x \mid E\}]\!]_{s,\pi,\mu} = \left\{ \begin{array}{ll} \{v \in [\![\textsc{Type}(x)]\!]_s \mid [\![E]\!]_{s,\pi,\mu \uplus [x/v]}\} & \text{if for all } v \in [\![\textsc{Type}(x)]\!]_s : \\ & [\![E]\!]_{s,\pi,\mu \uplus [x/v]} \neq \perp \\ \perp & \text{otherwise} \end{array} \right. \\
[\![E :: E']\!]_{s,\pi,\mu} = (\Lambda v \in [\![T]\!]_s, v' \in [\![[T]]\!]_s.\ v :: v')^{\perp}([\![E]\!]_{s,\pi,\mu}, [\![E']\!]_{s,\pi,\mu}) \text{ where } E : T \\
[\![\textbf{match } E_1 \textbf{ with } x :: y \textbf{ then } E_2 \textbf{ else } E_3]\!]_{s,\pi,\mu} = \\
\left\{ \begin{array}{ll} [\![E_2]\!]_{s,\pi,\mu \uplus [x/v, y/v']} & \text{if } [\![E_1]\!]_{s,\pi,\mu} = v :: v' \\ [\![E_3]\!]_{s,\pi,\mu} & \text{if } [\![E_1]\!]_{s,\pi,\mu} = \mathit{nil} \\ \perp & \text{otherwise} \end{array} \right. \\
[\![E_1 E_2]\!]_{s,\pi,\mu} = \left\{ \begin{array}{ll} f([\![E_2]\!]_{s,\pi,\mu}) & \text{if } f = [\![E_1]\!]_{s,\pi,\mu} \neq \perp \\ \perp & \text{otherwise} \end{array} \right. \\
[\![\lambda x.E]\!]_{s,\pi,\mu} = (\Lambda v \in [\![\textsc{Type}(x)]\!]_s.\ [\![E]\!]_{s,\pi,\mu \uplus [x/v]})^{\perp}
\end{array}
$$

Figure 6.13: Part 2 of denotational semantics of λXP, where s is a sequence of data trees, $\pi \in \text{nodes}(s)$, and μ to s-values is a well-typed variable assignment.

Proof. Proposition 3 show that the set V_s of all s-values is a CPO with least element $\perp$. It follows from Lemma 6 that all restrictions $([\![T_i]\!]_s, \sqsubset_{|[\![T_i]\!]_s \times [\![T_i]\!]_s})$ are CPO's too, whose least element is $\emptyset$. As a consequence, $(D, \sqsubset_D)$ is a CPO too whose least element is $(\emptyset,\ldots,\emptyset)$. □

6.3.4 Correctness

Proposition 6. *Let s be a sequence of trees, $\pi \in nodes(s)$, and μ a variable assignment to s-values. Let $\lambda x.E$ be a positive function. Then the function described by its denotation*

$$(\Lambda v \in [\![\textsc{Type}(x)]\!]_s.\ [\![E]\!]_{s,\pi,\mu \uplus [x/v]})^{\perp}$$

is Scott-continuous.

Proposition 7. *Let s be a sequence of trees, $\pi \in nodes(s)$, and μ a variable assignment to s-values. Let* **letrec** $x_1 = E_1, \ldots, x_n = E_n$ **in** E *be a recursive let expression, where all x_i occur in positive contexts in $E_1,\ldots,E_n$ and all E_i are positive functions. Then the function*

$$\Lambda v \in [\![T_1]\!]_s \times \ldots \times [\![T_n]\!]_s.([\![E_1]\!]_{s,\pi,\mu \uplus \nu(v)}, \ldots, [\![E_n]\!]_{s,\pi,\mu \uplus \nu(v)})$$

is Scott-continuous on the CPO *of Proposition 5, where $\nu = \Lambda(v_1,\ldots,v_n).[x_1/v_1] \uplus \ldots \uplus [x_n/v_n]$ and for all $1 \leq i \leq n$: $\textsc{Type}(x_i) = T_i$.*

$$\frac{E \triangleright V \to V'}{E \triangleright V \cup V'' \to V'} \qquad \frac{E \triangleright V \to V' \cup V''}{E \triangleright V \cup V'' \to V'}$$

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \uplus V' \to V''}{E_1 \wedge E_2 \triangleright V \to V' \cup V''} \qquad \frac{}{c \triangleright \emptyset \to \emptyset}\ \textit{true} \qquad \frac{}{x \triangleright \{x\} \to \emptyset}\ \textit{true}$$

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \to V'}{E_1 \vee E_2 \triangleright V \to V'} \qquad \frac{E \triangleright V \to V' \cup \{x\} \quad x \notin V'}{\exists x.E \triangleright V \to V'} \qquad \frac{E \triangleright V \to \emptyset}{\neg E \triangleright V \to \emptyset}$$

$$\frac{E \triangleright V \to V'}{A(E) \triangleright V \to V'} \qquad \frac{E \triangleright V \to V' \quad \text{E is navigational}}{\neg E \triangleright V \to V'} \qquad \frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to V'}{\mathbf{at}(E_1, E_2) \triangleright V \to V'}$$

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to V' \quad E_3 \triangleright V \to V'}{\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \triangleright V \to V'}$$

$$\frac{\forall 1 \leq i \leq n.\ E_i \triangleright V \uplus \{x_1, \ldots, x_n\} \to \emptyset \quad E \triangleright V \uplus \{x_1, \ldots, x_n\} \to V'}{\mathbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ E \triangleright V \to V'}$$

$$\frac{\forall 1 \leq i \leq n.\ E_i \triangleright V \to \emptyset}{(E_1, \ldots, E_n) \triangleright V \to \emptyset} \qquad \frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \uplus \{x_1, \ldots, x_n\} \to V'}{\mathbf{match}\ E_1\ \mathbf{with}\ (x_1, \ldots, x_n)\ \mathbf{in}\ E_2 \triangleright V \to V'}$$

$$\frac{E \triangleright V \to \{x\}}{\{x \mid E\} \triangleright V \to \emptyset}$$

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset}{E_1 :: E_2 \triangleright V \to \emptyset} \qquad \frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \uplus \{x, y\} \to V' \quad E_3 \triangleright V \to V'}{\mathbf{match}\ E_1\ \mathbf{with}\ x :: y\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \triangleright V \to V'}$$

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset}{E_1 E_2 \triangleright V \to \emptyset} \qquad \frac{E \triangleright V \uplus \{x\} \to \emptyset \quad x \notin V}{\lambda x.E \triangleright V \to \emptyset}$$

$$\frac{E \triangleright V \to \emptyset}{x = E \triangleright V \to \{x\}} \qquad \frac{E \triangleright V \to \emptyset}{x \in E \triangleright V \to \{x\}}$$

Figure 6.14: Mode judgements for well-typed λXP formulas.

6.4 Safety Restrictions

Let E be a λXP formula. A *mode* of E is a pair of variable subsets $V \to V'$ of input variables $V \subseteq \mathcal{V}$ whose values are known before evaluation of E, and output variables $V' \subseteq \mathcal{V}$ whose values will be produced during the evaluation of E. We write $E \triangleright V \to V'$ to say that E has mode $V \to V'$. Let $fV(E)$ be the set of free variables in E. We say that E is *well-moded*, if V' contains all free variables of E, except those variables in V, i.e. $fV(E) \setminus V \subseteq V'$.

Well-moded λXP formulas allow for efficient evaluation. Problems arise with existential quantification $\exists x.E$, where one would have to test E enumerating all possible values of $\textsc{Type}(x)$, and also in set constructions $\{x \mid E\}$ with the test of E. To avoid this we have designed a mode system for λXP in Figure 6.14 that guarantees well-modedness, such that well-moded λXP formulas can be efficiently evaluated.

For existential quantifications for example, we have mode rule:

$$\frac{E \triangleright V \to V' \cup \{x\} \quad x \notin V'}{\exists x.E \triangleright V \to V'}$$

It says that $\exists x.E$ is well-moded with input variables V and output variables V',

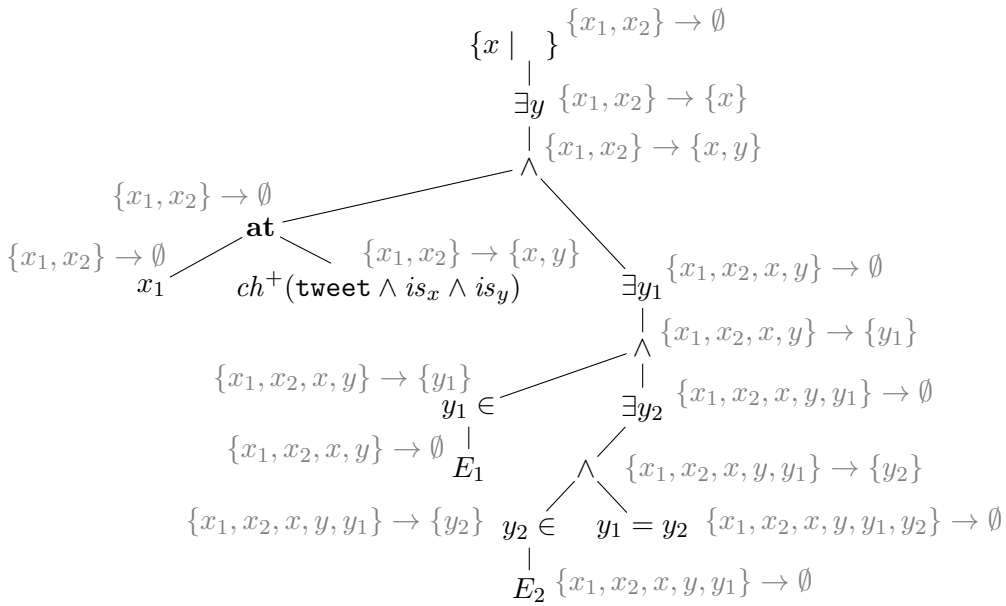


Figure 6.15: Mode-tree of query Q_1' from Section 10.1.1 with expanded shortcuts, and where $E_1 = \texttt{fn:data}(set2list(\{x' \mid \mathbf{at}(y, ch(\texttt{user\_name} \wedge is_{x'}))\}))$ and $E_2 = \texttt{fn:data}(set2list(\{x' \mid \mathbf{at}(x_2, ch^+(\texttt{students} \wedge ch(\texttt{name} \wedge is_{x'})))\}))$.

if expression E has mode $E \triangleright V \to V' \cup \{x\}$ producing values for variables x. An evaluation of $\exists x.E$ then does not have to guess all possible values for x, but evaluates E to generate a finite set of possible values for x.

For query Q_1' of Section 10.1.1 we give the *mode-tree* in Figure 6.15, i.e. where all terms in Q_1' are annotated by a mode, following the mode system. Notice that the mode system allows λXP to be differently well-moded, which is due to the two rules for submodes.

Note that for regular axis queries E^*, the mode system rules out E with mode $E \triangleright V \to V'$ where $V' \neq \emptyset$. E may contain free variables however, but then their values must be produced outside of E^*, e.g. when being used within another formula. This follows from the rule on lambda abstraction, but also from the rule on disjunctions.

Chapter 7

XPath to λXP

Contents

In this chapter we first define library functions that are required by the compiler from XPath 3.0 expressions to λXP terms that we present in a second step.

7.1 Instantiation of λXP

For the mapping from XPath expressions to λXP formulas, we instantiate λXP with parameters Σ and Δ for signatures, axes A, (label properties $\mathcal{L}$), constants $\mathcal{C}onst$, and parameter $\mathcal{A}(\mathcal{T}_{atom}, D, [\![.]\!]_., \leq_., \preceq_{atom})$ for atomic types.

The signatures are defined as discussed in Section 2.3.1:

$$\Sigma = \text{XML}_{types}$$
$$\Delta = \text{UTF8}$$

We allow the following axes:

$$A = \{ch, ch^+, ns^+, par, par^+, ps^+\}.$$

We define $\mathcal{C}onst$ to contain all constants that we define in Section 6.2.2, where we assume that arithmetic and comparison functions are defined as described by the specification for arithmetics and comparisons[1], and where the cast function is defined as presented in the specification[2]. We assume a function *sort* of type $[T] \to [T]$ for any equality type $T \in \mathcal{T}_{eq}$ that may implement any of the known sorting algorithms. Also by *NaN* we denote the double value of type `xs:double` that stands for *not a number*. Values *true* and *false* are booleans contained in $Bool \subseteq D$. Furthermore we add all XPath 3.0 built-in functions to $\mathcal{C}onst$, which can be mapped to λXP expressions. In Section 7.2 we only presented the translation of a small subset of the XPath 3.0 functions to λXP.

The parameter $\mathcal{A}$ for atomic types is defined as follows. The structure s is a sequence of data trees that are potentially available using XPath 3.0 functions `fn:doc`, `fn:unparsed-text`, and `fn:collections`. Function *root* from our constants then return for some the Uri that specifies a document the root node of the respective data tree, and similarly function *roots* takes a collection Uri for a collection of documents and returns a sequence of root nodes of respective data trees. Note that we model text files as data trees with a *doc*-root node, that has a single child of type *tx*, whose data values contains the content of the text file. $\mathcal{T}_{atom}$ is the finite set of atomic types that contains all atomic Xml types of Figure 4.1. The domain D contains all nodes of s and all atomic values of the Xml Schema Part 2 specification[3], i.e. the denotation of atomic types as discussed in Section 4.2.1. The denotation $[\![.]\!]_s$ maps the atomic types of $\mathcal{T}_{atom}$ to corresponding atomic values of $D \cup \text{nodes}(s)$. The subtype relation for atomic types was discussed in Section 4.4. Finally the order for atomic Xml types is defined in section "Operator Mapping"[4] of the XPath 3.0 specification.

7.2 Library functions

In this section we define library functions to translate XPath 3.0 expressions and functions. See also Section 6.2.6 for library functions of λXP expressions, such as functions *map*, *foldL*, and *append*, that are not purely related to the XPath 3.0 specification.

7.2.1 Access to Node Labels

In Figure 7.1 we present functions that operate on the label of nodes of Xml data trees. They use functions *labtype* and *labvalues* of Figure 6.2 that are defined according to the Xml node labels and thereby return the node type and node data values, respectively.

[1] https://www.w3.org/TR/xpath-30/#mapping
[2] https://www.w3.org/TR/xpath-functions-30/#casting
[3] https://www.w3.org/TR/xmlschema-2/
[4] https://www.w3.org/TR/xpath-30/#mapping

Function lab_{name} returns the value of node property "node-name" for element and attribute nodes. It verifies with function *labtype* whether its argument node is an element or an attribute node, and matches its data values according to function *labvalues* against a 6- or a 5-tuple, respectively, and returns its "node-name" property, which is stored at the second component. Function lab_{name} is defined only for element and attribute nodes, such that it raises an error for other node types. Notice that the value returned is a string that represents a QName, which is always composed of a namespace prefix and a local name that are separated by a colon ":".

Similar function $lab_{schema\text{-}type}$ returns the value of node property "schema-type" for element and attribute nodes. The default values for property "schema-type" in the absence of schema for elements is `xs:untyped` and for attributes it is `xs:untypedAtomic`. Function lab_{nilled} returns the value of node property "nilled" for element nodes, which is either string "true" or "false". Function lab_{target} returns the value of node property "target" for processing-instruction nodes. Function lab_{data} returns the value of node property "content" for text, comment, and processing-instruction nodes, and function lab_{value} returns the value of node property "string-value" for attribute nodes.

7.2.2 Lists of Nodes

The following functions transforms a list of nodes into a set of nodes, and it throws an error, if a non-node is encountered in the list

$nodelist2set =_{\text{df}}$
  letrec $nodelist2set' =$
    $\lambda z.$**match** z **with** $x :: y$ **then**
      if $type_{\text{TYPE}(x),\mathbf{node}}(x)$ **then** $\{x\} \cup nodelist2set'(y)$ **else raise** *error*
    else $\emptyset$
  in $nodelist2set'$

7.2.3 String Data Value of Nodes

The string-value of a node, which according to the specification is the concatenation of the string values of all its text-node descendants for element nodes, can be compiled into the λXP in Figure 7.2.

The `fn:string`[5] function returns the string-value of nodes, while its casts atomic values to string type `xs:string`.

$[\![\texttt{fn:string}]\!] =_{\text{df}}$
  $\lambda z.$**match** z **with** $x :: y$ **then**
    if $type_{\text{TYPE}(x),\mathbf{node}}(x)$ **then** $string\_value(x)$
    else if $type_{\text{TYPE}(x),\mathbf{xs:anyAtomicType}}(x)$ **then** $cast_{\text{TYPE}(x),\mathbf{string}}(x)$
    else raise *error*
  else ""

[5] http://www.w3.org/TR/xpath-functions-30/#func-string

$lab_{name} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = el$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_6)$ **in** x_2
else if $labtype(x) = att$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_5)$ **in** x_2
else raise *error*

$lab_{schema\text{-}type} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = el$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_6)$ **in** x_3
else if $labtype(x) = att$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_5)$ **in** x_3
else raise *error*

$lab_{nilled} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = el$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_6)$ **in** x_4
else raise *error*

$lab_{target} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = pi$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_3)$ **in** x_2
else raise *error*

$lab_{data} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = tx$ **then match** $labvalues(x)$ **with** (x_1) **in** x_1
else if $labtype(x) = com$ **then match** $labvalues(x)$ **with** (x_1) **in** x_1
else if $labtype(x) = pi$ **then match** $labvalues(x)$ **with** (x_1, x_2, x_3) **in** x_3
else raise *error*

$lab_{value} =_{\text{df}}$
$\lambda x.$ **if** $labtype(x) = att$ **then match** $labvalues(x)$ **with** $(x_1, \ldots, x_5)$ **in** x_1
else raise *error*

Figure 7.1: Access to Node Labels.

7.2.4 Atomization of Sequences

The typed-value of a node can be computed via its string-value:

$typed\_value =_{\text{df}}$
$\lambda x.$ **if** $type_{\text{TYPE}(x),\textbf{textnode}}(x) \vee type_{\text{TYPE}(x),\textbf{commentnode}}(x)$
$\vee type_{\text{TYPE}(x),\textbf{pinode}}(x)$ **then**
$lab_{data}(x)$
else
$cast_{\textbf{string},\text{TYPE}(lab_{schema\text{-}type}(x))}(string\_value(x))$

Function `fn:data`[6] that computes the atomization of a sequence, as discussed in Section 5.2.4.

$[\![\texttt{fn:data}]\!] =_{\text{df}}$

[6] http://www.w3.org/TR/xpath-functions/#func-data

$$
\begin{array}{l}
\mathit{string\_value} =_{\mathrm{df}} \\
\ \textbf{letrec}\ \mathit{string\_value}' = \\
\quad \lambda x.\ \textbf{if}\ \mathit{type}_{\mathrm{TYPE}(x),\textbf{textnode}}(x) \vee \mathit{type}_{\mathrm{TYPE}(x),\textbf{commentnode}}(x) \\
\qquad \vee \mathit{type}_{\mathrm{TYPE}(x),\textbf{pinode}}(x)\ \textbf{then} \\
\qquad\quad \mathit{lab}_{\mathit{data}}(x) \\
\qquad \textbf{else if}\ \mathit{type}_{\mathrm{TYPE}(x),\textbf{attribute}}(x) \\
\qquad\quad \mathit{lab}_{\mathit{value}}(x) \\
\qquad \textbf{else if}\ \mathit{type}_{\mathrm{TYPE}(x),\textbf{elementnode}}(x) \\
\qquad\quad \vee \mathit{type}_{\mathrm{TYPE}(x),\textbf{documentnode}}(x)\ \textbf{then} \\
\qquad\quad \mathit{listconcat} \\
\qquad\qquad (\mathit{map}\ \mathit{string\_value}' \\
\qquad\qquad\ \mathit{sort}(\mathit{set2list}(\\
\qquad\qquad\quad \{x' \mid \mathbf{at}(x, \mathit{ch}^{+}(\mathit{is}_{x'} \wedge \mathit{labtype}(x') = \mathit{tx})\}))) \\
\ \textbf{in}\ \mathit{string\_value}'
\end{array}
$$

Figure 7.2: Function *string_value* that computes strings of nodes.

$$
\begin{array}{l}
\lambda z.\mathit{map}\ (\lambda z'.\textbf{if}\ \mathit{type}_{\mathrm{TYPE}(z'),\textbf{xs:anyAtomicType}}(z')\ \textbf{then}\ z' \\
\qquad\qquad \textbf{else if}\ \mathit{type}_{\mathrm{TYPE}(z'),\textbf{node}}(z')\ \textbf{then}\ \mathit{typed\_value}(z')\ \textbf{else raise}\ \mathit{error})\ z
\end{array}
$$

7.2.5 Numerics

Function `fn:number`[7] accepts singleton lists only whose element is cast into a double of type `xs:double`. Otherwise the function returns the double value NaN.

$$
\begin{array}{l}
[\![\texttt{fn:number}]\!] =_{\mathrm{df}} \\
\quad \lambda z.\textbf{match}\ z\ \textbf{with}\ x :: y\ \textbf{then} \\
\qquad\quad \textbf{if}\ y = \mathit{nil}\ \textbf{then try}\ \mathit{cast}_{\mathrm{TYPE}(x),\textbf{xs:double}}(x)\ \textbf{catch}(e)\ \mathit{NaN}\ \textbf{else}\ \mathit{NaN} \\
\qquad \textbf{else}\ \mathit{NaN}
\end{array}
$$

Function `fn:count`[8] counts the number of elements in a sequence or list.

$$
\begin{array}{l}
[\![\texttt{fn:count}]\!] =_{\mathrm{df}} \\
\quad \textbf{letrec}\ \mathit{count} = \\
\qquad \lambda z.\ \textbf{match}\ z\ \textbf{with}\ x :: y\ \textbf{then}\ 1 + \mathit{count}(y)\ \textbf{else}\ 0 \\
\quad \textbf{in}\ \mathit{count}
\end{array}
$$

7.2.6 Booleans

Function `fn:boolean`[9] computes the effective boolean value that we discussed in Section 5.5.

$$
[\![\texttt{fn:boolean}]\!] =_{\mathrm{df}}
$$

[7] http://www.w3.org/TR/xpath-functions-30/#func-number
[8] http://www.w3.org/TR/xpath-functions-30/#func-count
[9] http://www.w3.org/TR/xpath-functions-30/#func-boolean

$\lambda z.$**match** z **with** $x :: y$ **then**
if $type_{\mathrm{TYPE}(x),\mathbf{node}}(x)$ **then**
true
else if $y = nil$ **then**
if $type_{\mathrm{TYPE}(x),\mathbf{bool}}(x)$ **then** x
else if $type_{\mathrm{TYPE}(x),\mathbf{string}}(x) \vee type_{\mathrm{TYPE}(x),\mathbf{xs:anyURI}}(x)$
$\vee\ type_{\mathrm{TYPE}(x),\mathbf{xs:untypedAtomic}}(x)$ **then** $\neg(x = \text{“”})$
else if $type_{\mathrm{TYPE}(x),\mathbf{num}}(x)$ **then**
if $x = 0 \vee x = NaN$ **then** *false* **else** *true*
else raise *error*(err:FORG0006)
else raise *error*(err:FORG0006)
else *false*

In the following we translate boolean functions `fn:true`, `fn:false`, and `fn:not`.

$[\![$`fn:true`$]\!] =_{\mathrm{df}}$
true

$[\![$`fn:false`$]\!] =_{\mathrm{df}}$
false

$[\![$`fn:not`$]\!] =_{\mathrm{df}}$
$\lambda z.\neg[\![$`fn:boolean`$]\!](z)$

7.2.7 Strings

Function `fn:concat`[10], for example, concatenates strings. It takes a list of atomic values, casts them to `xs:string` values, and concatenates the stings, treating them as lists of characters.

$[\![$`fn:concat`$]\!] =_{\mathrm{df}}$
$\lambda(z_1, \ldots, z_n).listconcat(map(cast_{\mathbf{xs:anyAtomicType},\mathbf{string}}, z_1 :: \ldots :: z_n :: nil))$

Other functions on strings[11] like `fn:contains` and `fn:normalize-space` can be implemented straightforwardly, by decomposition of lists of characters. Function `fn:contains` can be implemented by some function *contains* that tests whether a certain string contains another one, while `fn:normalize-space` removes from a string leading and trailing whitespace and reduces sequences of whitespace within the string to a single space character.

7.2.8 EQNames

EQNames are produced by nonterminal `EQName` (94) and are either **QNames** (104) or **URIQualifiedNames** (99). Their values are to be identified with lists of char-

[10] https://www.w3.org/TR/xpath-functions-30/#func-concat
[11] http://www.w3.org/TR/xpath-functions-30/#string-value-functions

acters, similar to strings.

QNames contain a local name (the actual label), and a namespace prefix. To have access to the individual parts, functions `fn:local-name-from-QName` and `fn:prefix-from-QName` exist. Namespace prefix and local name in **QNames** are separated by a colon ":", which makes it easy to implemented with an λXP expression that treats **QNames** as lists of characters.

URIQualifiedNames contain a namespace URI and a local name, of which the namespace URI is put in curly brackets which follows the local name. Similarly to before, we can think of functions *withinBrackets* and *afterBrackets* that return the namespace URI and local name. Also since we need to compare namespace prefixes to namespace URIs we assume a function *prefix* can returns the corresponding namespace prefix for some namespace URI.

The following function *testEQName* tests for some node z whether the data value of its property node-name (which represents a QName) matches **EQName** z' (which is either a QName or a URIQualifiedName).

$$
\begin{array}{l}
\mathit{testEQName} =_{\mathrm{df}} \\
\quad \lambda(z, z'). \\
\quad\quad \textbf{if } \mathit{type}_{\mathrm{TYPE}(z'),\textbf{xs:QName}}(z') \textbf{ then} \\
\quad\quad\quad \textbf{if } \mathit{contains}(z', \text{":"}) \textbf{ then} \\
\quad\quad\quad\quad \mathit{lab}_{name}(z) = z' \\
\quad\quad\quad \textbf{else } [\![\texttt{fn:local-name-from-QName}]\!](\mathit{lab}_{name}(z)) = z' \\
\quad\quad \textbf{else} \quad //\mathit{type}_{\mathrm{TYPE}(z'),\textbf{xs:URIQualifiedName}}(z') = \mathit{true} \\
\quad\quad\quad [\![\texttt{fn:local-name-from-QName}]\!](\mathit{lab}_{name}(z)) = \mathit{afterBrackets}(z') \wedge \\
\quad\quad\quad\quad [\![\texttt{fn:prefix-from-QName}]\!](\mathit{lab}_{name}(z)) = \mathit{prefix}(\mathit{withinBrackets}(z'))
\end{array}
$$

7.2.9 Positions

The *addPositionAndSize* function takes a list of elements and creates a new list with the same elements, but with information about the position of the element in the list and the size of the list. The result list stores triples that contain the original element, its index in the list, and the size of the list. E.g. *addPositionAndSize*("a" :: "b" :: *nil*) = ("a", 1, 2) :: ("b", 2, 2) :: *nil*.

$$
\begin{array}{l}
\mathit{addPositionAndSize} =_{\mathrm{df}} \\
\quad \lambda z.\ \textbf{letrec } \mathit{add\text{-}index} = \lambda(x_1, x_2, x_3). \\
\quad\quad\quad\quad \textbf{match } x_1 \textbf{ with } x :: y \textbf{ then} \\
\quad\quad\quad\quad\quad (x, x_2) :: \mathit{add\text{-}index}(y, x_2 + 1, x_3) \\
\quad\quad\quad\quad \textbf{else } \mathit{nil} \\
\quad\quad \textbf{in } \mathit{add\text{-}index}(z, 1, [\![\texttt{fn:count}]\!](z))
\end{array}
$$

Function *addReversePositionAndSize* similarly computes a list of triples, but stores reverse indices for elements. E.g. *addReversePositionAndSize*("a" :: "b" :: *nil*) = ("a", 2, 2) :: ("b", 1, 2) :: *nil*.

$$\mathit{addReversePositionAndSize} =_{\mathrm{df}}$$

$$\lambda z.\ \textbf{letrec}\ addReverseIndex = \lambda(x_1, x_2, x_3).$$
$$\textbf{match}\ x_1\ \textbf{with}\ x :: y\ \textbf{then}$$
$$(x, x_2) :: addReverseIndex(y, x_2 - 1, x_3)$$
$$\textbf{else}\ nil$$
$$\textbf{in}\ addReverseIndex(z, [\![\texttt{fn:count}]\!](z) + 1, [\![\texttt{fn:count}]\!](z))$$

7.2.10 Roots and Input Documents

The `fn:root`[12] function returns the root node of the data tree to which its argument node `$z` belongs: $[\![\texttt{fn:root(\$z)}]\!] = [\![\texttt{(\$z/ancestor-or-self::node())[1]}]\!]$.

In order to have access to input source documents we have added function *root* in λXP, for the translation of function `fn:doc`, which given an URI that represents a document, returns the root of the respective data tree:

$[\![\texttt{fn:doc}]\!] =_{\text{df}}$
root

Similar we have a function *roots* in λXP that mimics function `fn:collection` of XPATH 3.0, returning a list of root nodes of data trees for a collection URI:

$[\![\texttt{fn:collection}]\!] =_{\text{df}}$
roots

7.3 Compiling the XPath 3.0 Grammar to λXP

In this section we present the translation of a subset of XPATH 3.0 expressions to λXP expressions, while we refer to Appendix B for a complete translation of XPATH 3.0 to λXP that follows the *ordered* semantics of XSLT. We start by the translation of navigational XPATH 3.0 expressions relating to Chapter 3. Next we show how queries with positions with respect to Section 5.3 are translated. Finally, we present how ordering constraints can be relaxed within the *unordered* semantics of XQUERY, while otherwise we assume the standard *ordered* semantics of XSLT.

7.3.1 Navigational XPath Expressions

We present a translation for navigational XPATH 3.0 expressions, which we discussed in Chapter 3. The translation applies only for instances of XPATH 3.0 expressions `P` by the original XPATH grammar [Robie 2014b], i.e. those in Section 5.1.2, but only if query `P` can also be produced by the grammar of the navigational XPATH 3.0 fragment given in Section 3.2.2.

The translation of a navigational XPATH expression `P` that is an instance of nonterminal `N` produces a boolean λXP expression E. The translation is parameterized by some given boolean λXP formula F of properties to be verified at nodes

[12] http://www.w3.org/TR/xpath-functions-30/#func-root

that P navigates to, such as node label tests that we translate before the translation of axes steps. For the translation of navigational expressions we write

$$(\!|\texttt{N(P)}|\!)^F = E$$

The translation is initialized with $F = \mathit{true}$ such that the result of $(\!|\texttt{N(P)}|\!)^{\mathit{true}}$ is a boolean λXP formula that is satisfied by some data tree of which the XML document that it models satisfies the XPATH filter `[P]`. Otherwise F is set to is_x such that the result of $(\!|\texttt{N(P)}|\!)^{\mathit{is}_x}$ is satisfied by some data tree, if the value for variable x is amongst the nodes that XPATH query P selects. The nodes selected by expressions P then correspond to λXP expression

$$\{x \mid (\!|\texttt{N(P)}|\!)^{\mathit{is}_x}\}$$

We refer to the next section for an integration of the translation for navigational XPATH queries, that we present in the following, into a translation for arbitrary XPATH expressions.

Path Compositions. We start by the translation of expressions by nonterminal

(36) `RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*`

for the case that the expression uses the path operator "/":
$(\!|\texttt{RelativePathExpr(StepExpr(P)/StepExpr(P}')\texttt{)}|\!)^F =_{\text{df}}$
$\quad (\!|\texttt{StepExpr(P)}|\!)^{(\!|\texttt{StepExpr(P}')|\!)^F}$
The idea is that formula F is to be tested at nodes selected by `StepExpr(P)/StepExpr(P'`), so it reduces to translation $(\!|\texttt{StepExpr(P}')|\!)^F$, which is to be tested at nodes selected by `StepExpr(P)`. Navigational expressions by nonterminal

(37) `StepExpr ::= PostfixExpr | AxisStep`

are expressions by nonterminal `AxisStep` (38).

Filters. Expressions by nonterminal

(38) `AxisStep ::= (ReverseStep | ForwardStep) PredicateList`

require a translation of a forward or backward path step P, which is an instance of nonterminal `ReverseStep` or `ForwardStep`, respectively, together with a list of filters $\texttt{P}_i$ (by nonterminal `PredicatesList`) which are instances of nonterminal `Expr`.
$(\!|\texttt{AxisStep(P[Expr(P}_1\texttt{)]...[Expr(P}_n\texttt{)])}|\!)^F =_{\text{df}}$
$\quad (\!|\texttt{P}|\!)^{F \wedge (\!|\texttt{Expr(P}_1\texttt{)}|\!)^{\mathit{true}} \wedge \ldots \wedge (\!|\texttt{Expr(P}_n\texttt{)}|\!)^{\mathit{true}}}$
The idea is that F and all filters require evaluation at nodes by P.

Single Forward Path Steps.

(39) `ForwardStep::= (ForwardAxis NodeTest) | AbbrevForwardStep`

For nonterminal `ForwardStep` we are not interested in nonterminal `AbbrevForwardStep`, since considering only non-abbreviated XPath queries.

$(\!|\texttt{ForwardStep(ForwardAxis(P) NodeTest(P'))}|\!)^F =_{\text{df}}$
$(\!|\texttt{ForwardAxis(P)}|\!)^{F \wedge (\!|\texttt{NodeTest(P')}|\!)^{true}}$

F, together with the nodes label and kind tests by `NodeTest(P')` are tested at nodes by `P`.

Then we can translate the following expressions by nonterminal `ForwardAxis`(40) as follows, where *att* is a label property that accepts all labels of attribute nodes.

(40)
```
ForwardAxis::= ("child" "::") | ("descendant" "::") |
               ("attribute" "::") | ("self" "::") |
               ("descendant-or-self" "::") |
               ("following-sibling" "::") |
               ("following" "::") | ("namespace" "::")
```

$(\!|\texttt{ForwardAxis(child::)}|\!)^F =_{\text{df}}$
$ch(F \wedge \neg att)$
$(\!|\texttt{ForwardAxis(descendant::)}|\!)^F =_{\text{df}}$
$ch^+(F \wedge \neg att)$
$(\!|\texttt{ForwardAxis(attribute::)}|\!)^F =_{\text{df}}$
$ch(F \wedge att)$
$(\!|\texttt{ForwardAxis(self::)}|\!)^F =_{\text{df}}$
F
$(\!|\texttt{ForwardAxis(descendant-or-self::)}|\!)^F =_{\text{df}}$
$ch^*(F \wedge \neg att)$
$(\!|\texttt{ForwardAxis(following-sibling::)}|\!)^F =_{\text{df}}$
$ns^+(F \wedge \neg att)$
$(\!|\texttt{ForwardAxis(following::)}|\!)^F =_{\text{df}}$
$par^*(ns^+(ch^*(F \wedge \neg att)))$
$(\!|\texttt{ForwardAxis(namespace::)}|\!)^F =_{\text{df}}$
raise *error*

(41) `AbbrevForwardStep::= "@"?  NodeTest`

We do not consider abbreviated XPath queries.

Single Backward Path Steps. Concerned are the following three nonterminals, whose translation works similar to the translation of single forward path steps.

```
(42) ReverseStep       ::= (ReverseAxis NodeTest) | AbbrevReverseStep
(43) ReverseAxis       ::= ("parent" "::") | ("ancestor" "::") |
                           ("preceding-sibling" "::") |
                           ("preceding" "::") |
                           ("ancestor-or-self" "::")
(44) AbbrevReverseStep ::= ".."
```

$$(\!|\texttt{ReverseStep(ReverseAxis(P) NodeTest(P}')\texttt{)}|\!)^F =_{\text{df}} (\!|\texttt{ReverseAxis(P)}|\!)^{F \wedge (\!|\texttt{NodeTest(P}')|\!)^{true}}$$

$$(\!|\texttt{ReverseAxis(parent::)}|\!)^F =_{\text{df}} par(F \wedge \neg att)$$

$$(\!|\texttt{ReverseAxis(ancestor::)}|\!)^F =_{\text{df}} par^+(F \wedge \neg att)$$

$$(\!|\texttt{ReverseAxis(preceding-sibling::)}|\!)^F =_{\text{df}} ps^+(F \wedge att)$$

$$(\!|\texttt{ReverseAxis(preceding::)}|\!)^F =_{\text{df}} par^*(ps^+(ch^*(F \wedge \neg att)))$$

$$(\!|\texttt{ReverseAxis(ancestor-or-self::)}|\!)^F =_{\text{df}} par^+(F \wedge \neg att)$$

Node Labels and Kinds. Labels and kinds of nodes are verified by nonterminal

```
(45) NodeTest ::= KindTest | NameTest
```

They will be translated to label properties L, which are satisfied by all nodes whose label satisfied L.

An expression by nonterminal `NameTest` (46)

```
(46) NameTest ::= EQName | Wildcard
(94) EQName   ::= QName | URIQualifiedName
```

is a **QName** by nonterminal `QName` (104), for example, which is either a local name `P` or a composition `P'`:`P` of a namespace prefix `P'` and a local name `P`. We thereby obtain

$$(\!|\texttt{QName(P)}|\!)^{true} =_{\text{df}} P$$

$$(\!|\texttt{QName(P'}\texttt{:}\texttt{P)}|\!)^{true} =_{\text{df}} P \wedge P'$$

where we identify the result P of the translation with the label property that holds for all nodes whose local name is `P`, and label property P' holds for all nodes whose namespace prefix is `P'`.

We will not discuss in detail expressions by `Wildcard` (47). Furthermore, we only give a few expressions that check node kinds, as derived by nonterminal `KindTest` (71): Expressions `attribute()`, `element()`, and `text()` are satisfied by attribute, element, and text nodes, respectively. We translate these test expressions to corresponding label properties:

$(\!|\texttt{AttributeTest(attribute())}|\!)^{true} =_{\text{df}}$
$\quad att$
$(\!|\texttt{ElementTest(element())}|\!)^{true} =_{\text{df}}$
$\quad el$
$(\!|\texttt{TextTest(text())}|\!)^{true} =_{\text{df}}$
$\quad tx$

First-Order Connectives. The boolean connectives `and` and `or` by nonterminals

(16) `OrExpr ::= AndExpr ( "or" AndExpr )*`
(17) `AndExpr::= ComparisonExpr ( "and" ComparisonExpr )*`

and the XPath 3.0 functions `fn:true`, `false`, and `fn:not`, that may appear in filters of navigational XPath queries are translated straightforwardly as follows.
$(\!|\texttt{OrExpr(AndExpr(P) or AndExpr(P'))}|\!)^{true} =_{\text{df}}$
$\quad (\!|\texttt{AndExpr(P)}|\!)^{true} \vee (\!|\texttt{AndExpr(P')}|\!)^{true}$
$(\!|\texttt{AndExpr(ComparisonExpr(P) and ComparisonExpr(P'))}|\!)^{true} =_{\text{df}}$
$\quad (\!|\texttt{ComparisonExpr(P)}|\!)^{true} \wedge (\!|\texttt{ComparisonExpr(P')}|\!)^{true}$
$(\!|\texttt{fn:true()}|\!)^{true} =_{\text{df}}$
$\quad true$
$(\!|\texttt{fn:false()}|\!)^{true} =_{\text{df}}$
$\quad false$
$(\!|\texttt{fn:not(Expr(P))}|\!)^{true} =_{\text{df}}$
$\quad \neg(\!|\texttt{Expr(P)}|\!)^{true}$

Example. We present an example for the previously mentioned XPath query `descendant::a/child::b` that applies the presented translation rules:

$$\begin{aligned}
(\!|\texttt{descendant::a/child::b}|\!)^{is_x} &= (\!|\texttt{descendant::a}|\!)^{(\!|\texttt{child::b}|\!)^{is_x}} \\
&= (\!|\texttt{descendant::}|\!)^{(\!|\texttt{child::b}|\!)^{is_x} \wedge (\!|\texttt{a}|\!)^{true}} \\
&= ch^+((\!|\texttt{child::b}|\!)^{is_x} \wedge (\!|\texttt{a}|\!)^{true} \wedge \neg att) \\
&= ch^+((\!|\texttt{child::b}|\!)^{is_x} \wedge a \wedge \neg att) \\
&= ch^+((\!|\texttt{child::}|\!)^{is_x \wedge (\!|\texttt{b}|\!)^{true}} \wedge a \wedge \neg att) \\
&= ch^+(ch(is_x \wedge (\!|\texttt{b}|\!)^{true} \wedge \neg att) \wedge a \wedge \neg att) \\
&= ch^+(ch(is_x \wedge b \wedge \neg att) \wedge a \wedge \neg att)
\end{aligned}$$

7.3.2 Arbitrary XPath Expressions

We discuss how to translate arbitrary XPath expressions to λXP expressions. We translate an instance `P` of a nonterminal `N` of the XPath 3.0 grammar to some λXP expression E, and write

$$[\![\texttt{N(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} E.$$

The translation is parameterized by three variables $\bar{z}$, $\hat{z}$, and $\tilde{z}$, which is needed for a correct translation of positional queries. Those variables refer to the context item, context position, and context size, respectively. Before we explain in detail

the treatment of positional queries, we first discuss the integration of the translation for navigational queries that we presented before.

Navigational Expressions. The standard semantics claims that navigational XPath queries return duplicate-free sequences of nodes, which are to be sorted into document order, while the sorting can be relaxed within the unordered semantics of XQuery, as we describe later. We therefore reuse our translation for navigational expressions of before, and wrap it into a set, which we then transform into a list in an arbitrary order via library function *set2list*, while applying an additional sorting:

$$[\![\texttt{N(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} sort(set2list(\{x \mid (\!|\texttt{N(P)}|\!)^{is_x}\}))$$

Positional Queries. We discuss in the following how to evaluate filters and path expressions in general. For this let `P` and `P'` be subexpressions of filters `P[P']` and path expressions `P/P'` and `P!P'`, while we assume that expressions `P'` are not navigational and thus may contain positions, such as the context item expression "`.`", or function calls `fn:position()` or `fn:last()`, that we discussed in Section 5.3. Those expressions return the context item, position, and size, respectively. We recall, that the context item is the item currently being processed in sequence `P`, the context position is the position of the context item within the sequence, and the context size is the number of items in the processed sequence.

The question is how to translate those filters and path expressions and their subexpressions, since subexpressions `P'` reference items and positions of items in the result list by `P`, such that expressions `P'` cannot be translated without information of the evaluation of `P`. The idea is, that we translate `P'` as a function which requires the items, their positions, and the size of the list by `P` as input arguments, for which we translate `P` to a list of such triples, and filters and path expressions to function applications.

Context Item, Position, and Size. We thus translate the context item expression "`.`" by nonterminal `ContextItemExpr` (58), and function calls `fn:position()` and `fn:last()` to variables, whose values will be computed later, during evaluation.

$$[\![\texttt{ContextItemExpr(.)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \bar{z}$$
$$[\![\texttt{fn:position()}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \hat{z}$$
$$[\![\texttt{fn:last()}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \tilde{z}$$

Functions of Items, Position, and Size. Expressions `P'` of filters and path expressions, that contain context item expressions or functions calls, are translated to functions

$$\lambda(\bar{z},\hat{z},\tilde{z}).[\![\texttt{P'}]\!]_{\bar{z},\hat{z},\tilde{z}}.$$

This translation of subexpressions `P'` to functions allows to compute the filter or path expressions, when applied to triples of items, position, and size.

Lists of Triples of Items, Position, and Size. For items within a list, the information about its position within the list, and the size of the list, must be available directly, when evaluating filters and path expressions. We translate expressions `P` not to a list of items, but to a list of triples of items, positions, and list-size.

This is done via function *addPositionAndSize* presented in Section 7.2.9 of our function library. For the evaluation of filters and path expressions as mentioned above, we translate subexpressions `P` to $addPositionAndSize([\![\texttt{P}]\!]_{\bar{z},\hat{z},\tilde{z}})$. For example, this function takes the list ("a", "bc", 135) and outputs the list (("a", 1, 3), ("bc", 2, 3), (135, 3, 3)).

Evaluation of Filters. Filters `P[P']` by nonterminal

(38) `AxisStep::= (ReverseStep | ForwardStep) PredicateList`,

where `P` is an instance of nonterminal `ReverseStep` (42) or `ForwardStep` (39), and where `[P']` is a filter by nonterminal `PredicateList` (50), can be translated to function applications:

$$
\begin{array}{l}
[\![\texttt{P[P']}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\mathrm{df}} \\
\quad \textbf{let } \mathit{eval} = \lambda z.\textbf{match } z \textbf{ with } x :: y \textbf{ then} \\
\quad\quad \textbf{match } x \textbf{ with } (ci, cp, cs) \textbf{ in} \\
\quad\quad\quad \textbf{let } \mathit{filter} = \textbf{if } \mathit{type}_{\mathrm{TYPE}(ci),\textbf{node}}(ci) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \textbf{at}(ci, (\lambda(z^i, z^p, z^s).[\![\texttt{P'}]\!]_{z^i,z^p,z^s})(ci, cp, cs)) \\
\quad\quad\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad\quad\quad (\lambda(z^i, z^p, z^s).[\![\texttt{P'}]\!]_{z^i,z^p,z^s})(ci, cp, cs) \\
\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{if } \mathit{type}_{\mathrm{TYPE}(\mathit{filter}),\textbf{num}}(\mathit{filter}) \wedge \mathit{cast}_{\mathrm{TYPE}(\mathit{filter}),\textbf{int}}(\mathit{filter}) = cp \textbf{ then} \\
\quad\quad\quad\quad\quad\quad ci :: \mathit{eval}(y) \\
\quad\quad\quad\quad\quad \textbf{else if } [\![\texttt{fn:boolean}]\!](\mathit{filter}) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad ci :: \mathit{eval}(y) \\
\quad\quad\quad\quad\quad \textbf{else } \mathit{eval}(y) \\
\quad\quad\quad \textbf{else } \mathit{nil} \\
\quad\quad \textbf{in } \mathit{eval}(\mathit{addPositionAndSize}([\![\texttt{P}]\!]_{\bar{z},\hat{z},\tilde{z}}))
\end{array}
$$

For this, we define an evaluation function *eval* that inputs a list of triples $addPositionAndSize([\![\texttt{P}]\!]_{\bar{z},\hat{z},\tilde{z}})$. This function decomposes the list into triples x, which are made up of items ci, their position cp within the list, and the size cs of the list. Next, the function computes the function application $(\lambda(z^i, z^p, z^s).[\![\texttt{P'}]\!]_{z^i,z^p,z^s})(ci, cp, cs)$ which is evaluated from ci if ci is a node, and from $\bar{z}$ otherwise. The return value of the evaluation is stored in variable *filter*. When it is a numerical value, then the item ci is retained, if its position cp is equal to that value. Otherwise the **effective boolean value** of the value in *filter* is computed, where ci is retained if it evaluates to true.

Mapping Operator Expressions. Similarly we translate expressions that use the mapping operator "`!`" as by nonterminal

(34) `SimpleMapExpr::= PathExpr ("!" PathExpr)*`

$[\![\texttt{SimpleMapExpr(PathExpr(P) ! PathExpr(P')})]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let *apply* $= \lambda z.$**match** z **with** $x :: y$ **then match** x **with** (ci, cp, cs) **in**
if $type_{\text{TYPE}(ci),\textbf{node}}(ci)$ **then**
$\textbf{at}(ci, (\lambda(z^i, z^p, z^s).[\![\texttt{PathExpr(P')}]\!]_{z^i,z^p,z^s})(ci, cp, cs)) :: apply(y)$
else
$(\lambda(z^i, z^p, z^s).[\![\texttt{PathExpr(P')}]\!]_{z^i,z^p,z^s})(ci, cp, cs) :: apply(y)$
else *nil* **in**
listconcat $apply(addPositionAndSize([\![\texttt{PathExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

We define a function *apply* that performs the function applications. It gets as input a list of triples $addPositionAndSize([\![\texttt{PathExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}})$. For each triple the function application is performed evaluating `PathExpr(P')`, which returns a list of items. The overall result is thus a list of lists of items. Since sequences in XPath are flat, we flatten this result list by appending the lists that are contained, and we obtain a list of items.

Path Operator Expressions. Expressions that use the path operator "/" as by nonterminal

(36) `RelativePathExpr::= StepExpr (("/" | "//") StepExpr)*`

can be expressed using the mapping operator "!", see Section 5.4.2.

$[\![\texttt{RelativePathExpr(P/P')}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

```
[[RelativePathExpr(
let $R:= P!P' return
 if (every $r in $R satisfies $r instance of node()) then
   ($R union ())
 else if (every $r in $R satisfies not($r instance of node()))
   then $R
 else error()]]z̄,ẑ,z̃
```

7.3.3 Ordering Mode of XQuery

If ordering mode of XQuery is set to *ordered*, node sequences returned by path expressions, union, intersect, and except expressions, and the fn:id and fn:idref functions are to be sorted into document order. Otherwise the order of these return sequences is implementation-dependent. In the following we use some constant *mode* which is equal to the sort function *sort* if the ordering mode of XQuery is set to *ordered*, while it is the identity function *id*, when the ordering mode is set to *unordered*:

$$mode = \begin{cases} sort & \text{, ordering mode of XQuery is } ordered \\ id & \text{, ordering mode of XQuery is } unordered \end{cases}$$

Node Selecting Path Expressions. Depending on constant *mode* the result list of nodes for navigational expressions P is sorted or not:

$$[\![\texttt{N(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} mode(set2list(\{x \mid (\!|\texttt{N(P)}|\!)^{is_x}\}))$$

Union, Intersect, and Except Expressions. Expressions by nonterminals

```
(23) UnionExpr           ::= IntersectExceptExpr ( ("union" | "|")
                             IntersectExceptExpr )*
(24) IntersectExceptExpr ::= InstanceofExpr ( ("intersect" |
                             "except") InstanceofExpr )*
```

use operators `union`, `intersect`, and `except`, which operate on sequences of nodes only, see Section 5.7.1.

Notice that according to the specification, if one of the items in the sequences is not a node, a type error is raised, which we treat with library function *nodelist2set*.

$$\begin{aligned}&[\![\texttt{UnionExpr(IntersectExceptExpr(P}_1\texttt{) union ...}\\&\qquad\texttt{union IntersectExceptExpr(P}_n\texttt{))}]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}\\&\quad mode(set2list(nodelist2set([\![\texttt{IntersectExceptExpr(P}_1\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}}) \cup \ldots\\&\qquad\qquad \cup\ nodelist2set([\![\texttt{IntersectExceptExpr(P}_n\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})))\end{aligned}$$

Notice here, that translations $[\![\texttt{IntersectExceptExpr(P}_i\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ produce lists of nodes, which we transform to sets of nodes by function *nodelist2set*, whose set union is computed, and which afterwards is converted back to a list of nodes in an arbitrary order.

Expressions `IntersectExceptExpr` where $\texttt{op}_i \in \{\texttt{intersect}, \texttt{except}\}$ for all $1 \leq i \leq n$ translate similarly:

$$\begin{aligned}&[\![\texttt{IntersectExceptExpr(}\\&\quad\texttt{InstanceofExpr(P}_1\texttt{) op}_1\ \texttt{InstanceofExpr(P}_2\texttt{) ...}\\&\quad\texttt{InstanceofExpr(P}_n\texttt{) op}_n\ \texttt{InstanceofExpr(P}_{n+1}\texttt{))}]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}\\&\quad mode(set2list(\\&\quad nodelist2set([\![\texttt{InstanceofExpr(P}_1\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})\ \texttt{op}_1\\&\qquad nodelist2set([\![\texttt{InstanceofExpr(P}_2\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})\ \texttt{op}_2\\&\qquad\quad \ldots\ \texttt{op}_n\ nodelist2set([\![\texttt{InstanceofExpr(P}_{n+1}\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})))\end{aligned}$$

For Expressions. For expressions `for $x in P return P'` by nonterminal `ForExpr` in

```
(8)  ForExpr          ::= SimpleForClause "return" ExprSingle
(9)  SimpleForClause  ::= "for" SimpleForBinding
                          ("," SimpleForBinding)*
(10) SimpleForBinding ::= "$" VarName "in" ExprSingle
```

concatenate sequences of the return expression P′ in the order of the items in the binding sequence P if the ordering mode of XQuery is set to *ordered*:

$[\![$`ForExpr(for $VarName(y) in ExprSingle(P)`
`return ExprSingle(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

listconcat
(*map* ($\lambda[\![$`VarName(y)`$]\!]_{\bar{z},\hat{z},\tilde{z}}.[\![$`ExprSingle(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$) $[\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$)

There we perform a map on list $[\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ with function $\lambda[\![$`VarName(y)`$]\!]_{\bar{z},\hat{z},\tilde{z}}.[\![$`ExprSingle(P'`
while flattening the result list. For Example, query

```
for $x in (1, 33, 555) return ($x + 100, $x)
```

evaluates to list `(101,1,133,33,655,555)`. If the ordering mode were set to *unordered*, then the sequences by the return expression would not need to be concatenated in the order of the items of the binding sequence, but in an arbitrary order. For example, `(655,555,101,1,133,33)` is an admissible answer. However, the order of the items in the list by any evaluation of the return expression cannot be changed: `(1,33,101,133,555,655)` is *not* an admissible answer. In order to profit from this kind of order indifference, we allow the translation to add a call to XPath 3.0 function `fn:unordered` to the result of the evaluation of the binding sequence, which allows the sequence to be returned in an implement-dependent order, but only if the ordering mode is set to *unordered*:

$[\![$`ForExpr(for $VarName(y) in ExprSingle(P)`
`return ExprSingle(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

listconcat
(*map* ($\lambda[\![$`VarName(y)`$]\!]_{\bar{z},\hat{z},\tilde{z}}.[\![$`ExprSingle(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$)
$[\![$`fn:unordered`$]\!]([\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$)

Notice that function `fn:unordered` only relaxes the order for its immediate operand, while the ordering mode *unordered* of XQuery applies to all nested expressions.

Part II

Early Query Answering for Navigational XPath

Chapter 8

Early Nested Word Automata

Contents

In this chapter, we propose a new algorithm that approximates earliest query answering for navigational XPath queries. Our *early* query answering algorithms is based on early nested word automata that we introduce that are obtained from Fxp, the forward navigational fragment of λXP. We test our algorithms in detail on the usual benchmarks in an experimental section with promising performance.

8.1 Introduction

We present an approximation of earliest query answering for navigational XPath queries. One objective is to improve on previous approximations, in order to support earliest rejection for XPath queries with negation, such as for instance:

```
//tweet[not(pub/text()='Springer')][contains(text(),'Lille')]
```

When applied to an Xml document for an electronic library, as below, all books published from `Springer` can be rejected once its publisher was read:

```
<lib>...<book>...<pub> Springer </pub>
              ...<content>...Lille...</content>...</book>...</lib>
```

Spex's algorithm on the basis of transducer networks [Olteanu 2007], however, will check for all books from Springer whether they contain the string `Lille` and detect rejection only when the closing tag `</book>` is met. This requires unnecessary buffering space.

We provide an approximation of the earliest query answering algorithm for queries defined by Nwa [Gauwin 2009b, Madhusudan 2009], while removing the assumption of determinism imposed there. The main idea to gain efficiency is that selection and rejection should depend only on the current state of an Nwa but not on its current stack. Therefore, we propose *early nested word automata* (eNwas) that are Nwas with two kinds of distinguished states: rejection states and selection states. Selection states are final and must always remain final, so that a nested word can be accepted, once one of its prefixes reaches a selection state. Symmetrically, rejection states can never reach a final state, so that a nested word can be rejected, once all non-blocking runs on a prefix reach a rejection state. We then present a new streaming algorithm for answering eNwa queries in an early manner. The basic idea is to run the eNwa for all possible candidates while determinizing on-the-fly, so that one can see easily whether all non-blocking runs of the nondeterministic automaton reach a rejection state, or whether one of them is selecting. The second idea is to share the stacks and states of runs of buffered candidates in the same state, so that the running time does not depend on the number of buffered candidates, but only on the number of states of the deterministic automaton discovered during the on-the-fly determinization. Our streaming algorithm with stack-and-state sharing for answering eNwas queries is original and nontrivial. It enables tight upper bounds for time and space complexity that we prove (Theorem 3).

Furthermore, we show how to compile XPath expressions to small eNwa descriptors defining the same query. These descriptors allow to represent eNwas with large finite alphabets in a succinct manner, by replacing labels in eNwa rules by label descriptors. The label descriptor $\neg a$, for instance, stands for the set of all finitely many labels different from a. The target of our XPath-compiler are thus eNwa descriptors. For instance, the eNwa descriptors that our XPath compiler obtains for the XPath expressions P_n = `child::`a_1`/child::`a_2`/.../child::`a_n is

of size $O(n)$, while the described eNWA is of size $O(n^2)$. The latter has n states each of which has n transitions, in order to accept children with all possible letters $a_1, \ldots, a_n$. We will prove a tight time bound for our compiler (Theorem 2). It implies the same bound on the size of the generated eNWA descriptors, and in particular that the eNWA descriptors for any XPATH expressions without filters, unions, and with no other axes than *child* axes (such as P_n for instance) can be compiled in time $O(n)$. This improves on the previous compiler to dNWAs from [Gauwin 2011a], which required time $O(n^4)$ for P_n. The main idea of the compiler is to adapt the previous translation to dNWAs, so that it produces descriptors of eNWAs while distinguishing selection and rejection states. We maintain pseudo-completeness (no run can ever block) as an invariant, so that we can compile negations efficiently in the deterministic case. Otherwise, we treat negation based on eNWA determinization after the instantiation of the eNWA descriptors, even though this is costly in theory and often unfeasible in practice. The XPATH operators introducing nondeterminism are recursive axes such as *descendant*, *following* and *following-sibling*, disjunctions of filters and unions of paths, and also expressions with *child* axes such as `child::a[following::b]` where the subexpression cannot be decided when closing the child. It should also be noticed that the compilation of forward axis requires a more complex treatment of stack symbols during the eNWA construction, which leads to a more tedious correctness statement for our compiler.

We have implementated our algorithms to answer navigational XPATH queries in the QUIXPATH system. It improves on all other tools in coverage with 37% of the XPATHMARK benchmark (the previous best is SPEX with 22%), and also outperforms all of them in time efficiency with the exception of GCX, which runs slightly quicker on few queries, and slightly slower on others. Our approximation of earliest query answering turns out to be tight in practice, in that all supported queries of the XPATHMARK are treated in an earliest manner. Our algorithms are not earliest on particular XPATH queries with valid or unsatisfiable filters, of which there are two in the XPATHMARK. Although they are not supported by the algorithms presented in this chapter (since they use recursive axes below a negation), they are supported by our algorithm in Chapter 10, which answer them in a non-earliest manner. It is shown in follow-up work [Lick 2013] that our approximation is also tight in theory, in that it is exact for all positive XPath queries without valid or unsatisfiable subfilters. Note that there is still a gap between the tightness results in practice presented here, and those in the theory of the follow-up paper, in that some of the supported queries of the XPATHMARK use negation.

8.2 Automata Queries

We introduce how nested word automata are used to define node selection queries on XML data trees. We also introduce what it means to answer node selection queries on data trees in streaming mode in an earliest manner.

8.2.1 Tuple Selection

We consider tuple selection queries on marked trees, which select n-tuples of nodes of the marked tree.

Definition 9. *A tuple selection (forward) query over alphabets Σ and Δ is a function P^n that maps all marked trees (t, π) over Σ and Δ to some subset $P^n(t, \pi)$ of n-tuples of nodes $(\pi_1, \ldots, \pi_n)$ of t opened later or equal to π, i.e.,*

$$P^n(t, \pi) \subseteq \{(\pi_1, \ldots, \pi_n) \mid \textit{for all } 1 \leq i \leq n. \textit{fut}^t(\pi, \pi_i)\}.$$

These kinds of tuple selection queries cannot select nodes opened before π. We impose this restriction since we are interested in XPath queries with forward axes only in this chapter. From a streaming perspective, this means that queries of the above type concern only the tree suffix $\textit{suff}(t, \pi)$ defined by a marked tree (t, π). An example for a binary query is to select all pairs of nodes of a-descendant with their b-children.

Even though tuple selection queries are needed for answering general XPath queries over Xml streams, see Chapter 10, we consider mostly monadic queries for the ease of presentation.

8.2.2 Monadic Queries

Monadic queries P are tuple selection queries that select tuples of size 1:

$$P(t, \pi) \subseteq \{\pi' \mid \textit{fut}^t(\pi, \pi')\}$$

The XPath query `following-sibling::b`, for instance, will select the nodes 2 and 3 of the marked tree $(a(b, b, b), 1)$, since the b-node 1 of $a(b, b, b)$ has the next sibling 2 with label b, which in turn has the next sibling 3 again labeled by b. This is an example of a query that does not make much sense when started at the root of a tree, since the root never has any following sibling. For this reason, we apply queries to marked trees and let the query start at the marked node.

8.2.3 Queries by Automata

We will use Nwas to define monadic queries (as usual for showing that tree automata capture monadic second-order (Mso) queries). The idea is that an Nwa should only test whether a candidate node is selected by the query on a given tree suffix, but not generate the candidate by itself. We fix a single variable x for annotation and set the label alphabet of such Nwas to $\{a, a^x \mid a \in \Sigma\}$. Letters a^x are called annotated (or "starred" in the terminology of [Madhusudan 2009]) while letters a are not. Then, a unique candidate node is assumed to be annotated on the input tree suffix by some external process. A monadic query P on marked trees can be defined by any Nwa that recognizes the set of variants of $\textit{suff}(t, \pi)$, in which the label of a single selected node in $P(t, \pi)$ is annotated by x.

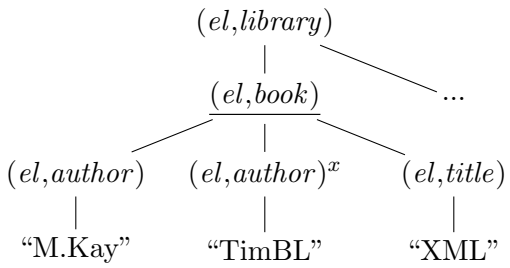


Figure 8.1: An example of a data tree for a library, in which the first `book`-element is marked. The second `auth`-child of the marked node is annotated by variable x.

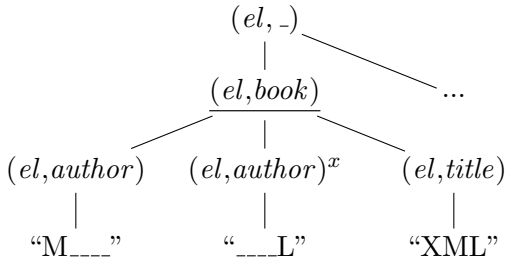


Figure 8.2: A data tree with finite signature, obtained from the data tree in Figure 8.1 via some relabling function that anonymizes letters not occurring in `book[starts-with(title,'XML')]/auth`.

Example. An example for a marked data tree of a library with an annotated label is given in Figure 8.1. There only `auth`-elements and `title`-elements carry a data value. The first `book`-element is chosen as the marked node. The second `auth`-child of the marked `book`-element node is annotated by the variable x. The XPath query

```
book[starts-with(title,'XML')]/auth
```

selects the annotated node when applied to the marked library (without the annotation), since the marked node is a `book`-element, whose title starts with “XML” and since the annotated node is an `auth`-child of the marked node.

Whether the above XPath query applied to a marked tree can select a given node, can be verified as follows. First we annotate the given node with x as in Figure 8.1, second we anonymize all symbols not occurring in the query by substitution with _, as in Figure 8.2, so that the signatures become finite, and third we run the deterministic Nwa in Figure 8.3 on the marked, annotated, and anonymized tree in Figure 8.2.

A successful run on the annotated marked tree from Figure 8.2 is depicted in Figure 8.4. This S-run starts at the marked `book` node, with the start stack $S = \gamma$ containing a single element, since the marked node has a single ancestor.

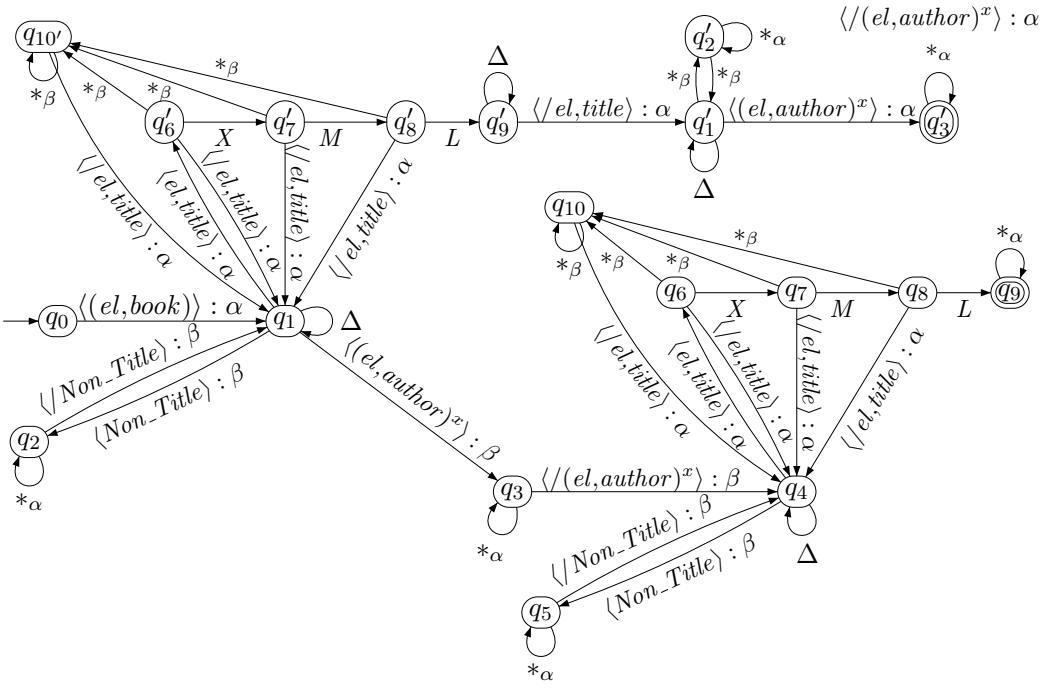


Figure 8.3: An NWA for XPATH query `book[starts-with(title,'XML')]/auth` selecting all authors of all books of a library whose title starts with "XML". It can be run on any marked library while starting at its marked node, under the assumption that there is an arbitrary but unique node annotated by x. This is a candidate node for which selection is to be verified by the NWA. The finite signatures, obtained by anonymization of all letters that do not belong to the query to $\_$, are $\Sigma = \{a, a^x \mid a \in \Sigma'\}$ where $\Sigma' = \{(el,book), (el,title), (el,author), (el,\_)\}$ and $\Delta = \{L, M, X, \_\}$. As shortcuts, we use the sets of transitions $*_\alpha = \{\langle a\rangle : \alpha,\ \langle/a\rangle : \alpha \mid a \in \Sigma'\} \cup \Delta$ and $*_\beta$ defined analogously to $*_\alpha$, and the set of tags $Non\_Title = \Sigma' \setminus \{(el,title)\}$.

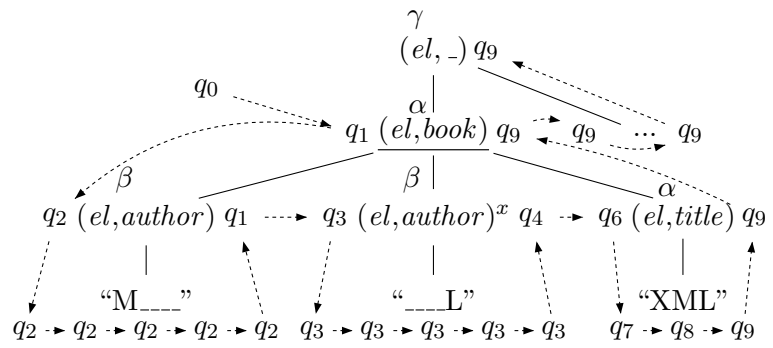


Figure 8.4: A successful run of the NWA of Figure 8.3 on the annotated library from Figure 8.2, in which the first `book` element is chosen as the marked node. Here, the stack $S = \gamma$ provides a stack symbol for the unique ancestor of the marked node.

8.2.4 Earliest Query Answering

Let P be a query, (t, π) a marked tree, π' a node such that $fut^t(\pi, \pi')$ is true, and e an event of $suff(t, \pi')$. We call π' *safe for selection at* e if $\pi' \in P(t', \pi)$ for every t' being a continuation of t beyond e, i.e., such that the prefixes of the linearizations of t and t' until event e are equal. We call π' *safe for rejection at* e if $\pi \notin P(t', \pi)$ for every t' such that t' is a possible continuation of t beyond e. We call π *alive at* e if it is neither safe for selection nor rejection at e. An earliest query answering (EQA) algorithm outputs selected nodes at the earliest event when they become safe for selection, and discards rejected nodes at the earliest event when they become safe for rejection. Indeed, an EQA algorithm buffers only alive nodes. The problem to decide the aliveness of a node is EXPTIME-hard for queries defined by NWAs [Gauwin 2009b]. For dNWAs it can be reduced to the reachability problem of pushdown machines which is in cubic time [Gauwin 2011a]. This, however, is too much in practice with NWAs of more than 50 states, 50 stack symbols, and $4 * 50^2 = 10.000$ transition rules, so that the time costs are in the order of magnitude of $10.000^3 = 10^{12}$.

8.3 Early Nested Word Automata

We will introduce early NWAs for approximating earliest query answering for NWAs with high time efficiency. The idea is to avoid reachability problems of pushdown machines, by enriching NWAs with selection and rejection states[1], so that aliveness can be approximated by inspecting states, independently of the stack. As we will see in Section 8.5, we can indeed distinguish appropriate selection and rejection states when compiling XPATH queries to ENWAs descriptors.

A subset Q' of states of an NWA A is called an *attractor* if any run of A that reaches a state of Q' can always be continued and must always stay in a state of Q'. Whether runs reaching Q' can always be continued is not that easy to decide syntactically, since it requires to decide accessibility questions for pushdown automata. In practice, however, we will only consider attractors that consists of a single state, which can loop in itself with all possible transitions.

Definition 10. *An* early nested word automaton *(*ENWA*) is a triple* $E = (A, Q_S, Q_R)$ *where* A *is an* NWA*,* Q_S *is an attractor of* A *of final states called selection states, and* Q_R *an attractor of non-final states called rejection states.*

In the example NWA in Figure 8.3, we can define $Q_S = \{q_9\}$ and $Q_R = \emptyset$. We could add a sink state to the automaton and to the set of rejection states. Also all selection states can be merged into a single state, and all rejection states can be deleted or merged into a single sink, if one wants to preserve pseudo-completeness (no run can ever block), as needed for efficient complementation in the deterministic case.

[1] The semantics of our selection states is identical with the semantics of final states in the acceptance condition for NWAs in [Alur 2009], but should not be confused in general.

An eNwa defines the same language or query as the underlying Nwa. Let us consider an eNwa E defining a monadic query and a data tree with some annotated node π. Clearly, whenever *some* run of E on this annotated tree reaches a selection state then π is safe for selection. By definition of attractors, this run can always be continued until the end of the stream while staying in selection states and thus in final states. In analogy, whenever *all* runs of E reach a rejection state, then π is safe for rejection, since none of the many possible runs can ever escape from the rejection states by definition of attractors, so none of them can be successful. For finding the first event, where all runs of E either reach a rejection state or block, it is advantageous to assume that the underlying Nwa is deterministic. In this case, if some run reaches a rejection state or blocks, we can conclude that all of them do, as there is at most one run.

We call an eNwa deterministic if the underlying Nwa is. We next argue that the determinization procedure for Nwas, see Section 2.4.4, can be lifted to eNwas. Let $E = (A, Q_S, Q_R)$ be an eNwa and A' the determinization of Nwa A. The states of A' are sets of pairs of states of A', and not just sets of states of A, in contrast to more traditional classes of automata. We define the deterministic eNwa $E' = (A', {Q_S}', {Q_R}')$ such that ${Q_S}'$ contains all sets of pairs of states of A, such that the second component of *some* pair belongs to Q_S, while ${Q_R}'$ contains all sets of pairs of states of A, for which *all* pairs have their second component in Q_R.

Lemma 7. *Let $E = (A, Q_S, Q_R)$ be an* eNwa *and $E' = (A', {Q_S}', {Q_R}')$ be the deterministic* eNwa *obtained from the above determinization procedure. For any event e of the stream of a tree t, there exists a run of E going into Q_S at event e if and only if there is a run of E' going into ${Q_S}'$ at e. Likewise all runs of E go into Q_R at event e iff all runs of E' go into ${Q_R}'$ at e.*

This means that eNwa determinization preserves early selection and rejection. Intuitively, the reason is as follows. If we ignore the first components of the state pairs, then the run of automaton E' on t always reaches the set of states reached by E on t. Hence, whenever some run of E on t goes into a selection state q, the unique run of E' goes into a set of states that contains q and is thus selecting for E'. And whenever all runs of E on t reach a rejection state (or block), then the unique run of E' on t reaches the set of all these rejection states, which is rejecting for E'.

8.4 Fxp

By Fxp we denote the forward navigational fragment of λXP extended with label formulas, and comparisons to static string data values, whose abstract syntax is given in Figure 8.5. It is parameterized by some alphabet Σ, an alphabet Δ for strings, a set of variables $\mathcal{V}$, and a set of label properties $\mathcal{L}$, where a label property $L \in \mathcal{L}$ verifies whether a property holds about some label in Σ, i.e. $L : \Sigma \to \mathbb{B}$. Formulas are constructed from the single atomic formula *true*, the usual boolean

$$
\begin{array}{lrcl}
\text{Formulas} & F & ::= & \mathrm{F} \wedge \mathrm{F} \mid \mathrm{F} \vee \mathrm{F} \mid \neg F \mid \mathit{true} \mid A(F) \mid B(F) \mid O_w \\
\text{Forward Axes} & A & ::= & \mathit{ch} \mid \mathit{ch}^+ \mid \mathit{ns}^+ \mid \mathit{fo} \\
\text{Label formulas} & B & ::= & \mathit{is}_x \mid L \mid B \& B \\
\text{Comparisons} & O & ::= & \mathit{equals} \mid \mathit{contains} \mid \mathit{starts\text{-}with} \mid \mathit{ends\text{-}with}
\end{array}
$$

Figure 8.5: Abstract syntax of Fxp where $x \in \mathcal{V}$ is a variable, $L \in \mathcal{L}$ a label predicate on Σ, and $w \in \Delta^*$ a string data value.

operators, forward axes A, label formulas $B(F)$, where B imposes a set of label properties $L \in \mathcal{L}$ and a set of variables annotations $x \in \mathcal{V}$, and string comparisons equals_w, $\mathit{contains}_w$, $\mathit{starts\text{-}with}_w$, and $\mathit{ends\text{-}with}_w$ where $w \in \Delta^*$.

For some data tree t the label properties $L \in \mathcal{L}$ are used as unary relation symbols and interpreted as the set L^t of nodes of t whose label satisfies L. The unary relations $\mathit{contains}^t_w$, equals^t_w, $\mathit{starts\text{-}with}^t_w$ and $\mathit{ends\text{-}with}^t_w$ for words $w \in \Delta^*$ are satisfied by all nodes π of t whose data values satisfy the respective relation to w.

Navigational XPath queries are compiled to Fxp formulas as described in Section 7.3.1. The XPath query `book[starts-with(title,'XML')]/auth`, for example, will be compiled to the following Fxp formula with one free variable x:

$$\mathit{el}\&\mathit{book}(\mathit{ch}(\mathit{el}\&\mathit{title}(\mathit{starts\text{-}with}_{XML}))) \wedge \mathit{ch}(\mathit{el}\&\mathit{author}\&\mathit{is}_x(\mathit{true}))))))$$

In this example, we rely on the following label properties: the label property *el* stands for all node labels (el, a) with $a \in \Sigma$, the label book for all (T, book) where T is one of the Xml types, and similarly *title* and *author* test whether a node label has the respective Xml tag. In addition, we use relations symbols *ch* and $\mathit{starts\text{-}with}_{XML}$ for talking about the corresponding relations of data trees.

We define the *conjunction width* $w(F)$ as the number of conjunction operators $\wedge$ in F. The conjunction width of an Fxp formula will be relevant for the complexity analysis of the automata construction in Section 8.5. Note that conjunctions in label formulas $B\&B'$ are not counted. Furthermore, $B(F)$ can be rewritten equivalently with a conjunction $B \wedge F$, but this would increase the conjunction width.

We recall the formal semantics of Fxp in Figure 8.6 for a single data tree t, while not considering program errors. A formula F is evaluated to a Boolean with respect to a given marked data tree (t, π) and a variable assignment μ that maps all variables of F to nodes of t opened later than π. The value of a formula F is the Boolean $[\![F]\!]_{t,\pi,\mu}$ defined in Figure 8.6. As usual we will write $F \models F'$ if all models of F are also models of F', i.e. if $[\![F]\!]_{t,\pi,\mu}$ is true then also $[\![F']\!]_{t,\pi,\mu}$.

Any formula F with one free variable x defines a monadic query P on marked trees such that $P(t, \pi) = \{\mu(x) \mid [\![F]\!]_{t,\pi,\mu} = \mathit{true}\}$. For compiling XPath expressions to the fragment sketched above, we need only such Fxp formulas. For the general case, as treated in the remainder of this chapter, formulas with n free

$$\begin{array}{ll}
[\![F_1 \wedge F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \wedge [\![F_2]\!]_{t,\pi,\mu} & [\![B(F)]\!]_{t,\pi,\mu} \Leftrightarrow [\![B]\!]_{t,\pi',\mu} \wedge [\![F]\!]_{t,\pi',\mu} \\
[\![F_1 \vee F_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![F_1]\!]_{t,\pi,\mu} \vee [\![F_2]\!]_{t,\pi,\mu} & [\![is_x]\!]_{t,\pi,\mu} \Leftrightarrow \pi = \mu(x) \\
[\![\neg F]\!]_{t,\pi,\mu} \Leftrightarrow \neg\, [\![F]\!]_{t,\pi,\mu} & [\![L]\!]_{t,\pi,\mu} \Leftrightarrow \pi \in L^t \\
[\![true]\!]_{t,\pi,\mu} \Leftrightarrow true & [\![B_1 \& B_2]\!]_{t,\pi,\mu} \Leftrightarrow [\![B_1]\!]_{t,\pi',\mu} \wedge [\![B_2]\!]_{t,\pi',\mu} \\
[\![A(F)]\!]_{t,\pi,\mu} \Leftrightarrow \exists \pi'.\ A^t(\pi,\pi') \wedge [\![F]\!]_{t,\pi',\mu} & [\![O_w]\!]_{t,\pi,\mu} \Leftrightarrow \pi \in O^t_w
\end{array}$$

Figure 8.6: Semantics of FXP formulas F for an XML data tree t with node π and variable assignment μ to nodes of t.

variables will be essential. Since the general case does not raise any additional difficulties, we will not impose any restriction on the number of variables.

8.5 Compiler from FXP to Early Nested Word Automata

We have already seen in the previous examples of NWAs for XPATH expressions[2] that the number of similar transitions for different labels may become huge. Furthermore, the alphabet of NWAs are exponential in the number of variables, which may become huge in the n-ary case. Therefore, we will compile FXP formulas into compact descriptors of ENWAs, in which labels are replaced by label descriptors.

8.5.1 ENWA Descriptors

Let $\mathcal{V}_0 \subseteq \mathcal{V}$ be a finite set of variables and $\mathcal{L}_0 \subseteq \mathcal{L}$ a finite set of label predicates over Σ, for instance those used by some fixed FXP formula.

A descriptor D_{lab} for a letter in Σ is an expression $E_1 \& \ldots \& E_n$ where all E_i belong to $\{L, \neg L \mid L \in \mathcal{L}_0\}$ and $n \geq 0$. We write $D_{lab} = true$ if $n = 0$. We define the denotation $[\![D_{lab}]\!] \subseteq \Sigma$ as follows:

$$[\![L]\!] = L \quad \text{and} \quad [\![\neg L]\!] = \Sigma \setminus L \quad \text{and} \quad [\![E_1 \& \ldots \& E_n]\!] = [\![E_1]\!] \cap \ldots \cap [\![E_n]\!].$$

A descriptor D_{var} for a subset of $\mathcal{V}_0$ is a pair (V, V') of subsets of $\mathcal{V}_0$. Its denotation is defined as follows:

$$[\![(V_1, V_2)]\!] = \{V \mid V_1 \subseteq V \subseteq (\mathcal{V} \setminus V_2)\}$$

We define the conjunction of two such descriptors $(V_1, V_1')\&(V_2, V_2')$ as $(V_1 \cup V_2, V_1' \cup V_2')$. Clearly, $[\![(V_1, V_1')\&(V_2, V_2')]\!]$ is equal to $[\![(V_1, V_1')]\!] \cap [\![(V_2, V_2')]\!]$.

A descriptor D_{tup} for a triple in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ is a triple (D_1, D_2, D_3) consisting of a descriptor D_1 for a letter in Σ, and descriptors D_2 and D_3 for subsets of $\mathcal{V}_0$. The denotation is $[\![(D_1, D_2, D_3)]\!] = [\![D_1]\!] \times [\![D_2]\!] \times [\![D_3]\!]$.

[2] See Figure 8.3 and also the expressions P_n in the introduction.

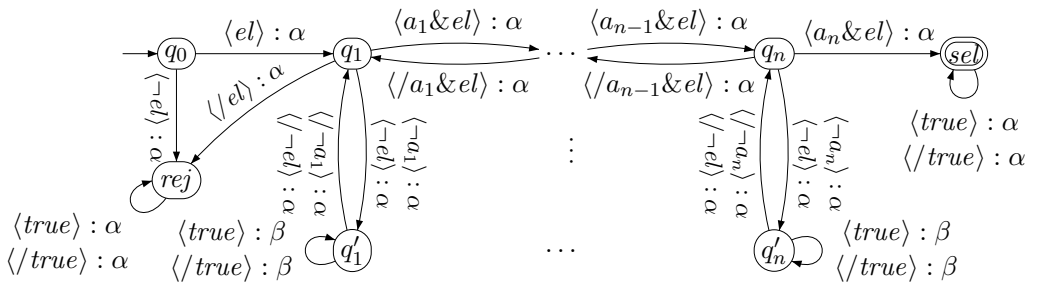


Figure 8.7: A descriptor of an eNWA for XPATH filter `[child::`a_1`/child::`a_2`/.../child::`a_n`]` with selection state *sel* and rejection state *rej* (without transitions for texts and other XML types for simplicity).

A descriptor of an eNWA is an eNWA itself, whose alphabets of opening and closing tags contain descriptors. An eNWA descriptor thus uses opening and closing tags $\langle D\rangle$ and $\langle /D\rangle$ where D is a descriptor of a label in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ rather than the label itself. The intuition for why we consider labels in $\Sigma \times 2^{\mathcal{V}_0} \times 2^{\mathcal{V}_0}$ is that we construct automata, who need to check a label in Σ, an annotation in $2^{\mathcal{V}_0}$, and who needs to check the possibility for annotations in $2^{\mathcal{V}_0}$ of future nodes of the data tree. This will become clearer in the following. The eNWA described is obtained from an eNWA descriptor by instantiating all occurrences of letters D in transition rules by all possible values of $[\![D]\!]$. We also need similar descriptors for letters in Δ but omit the details here. Note that it is possible that a rule is described twice by an eNWA descriptor, while the described automaton is still deterministic.

In Figure 8.7, we illustrate the eNWA descriptor for XPATH filter `[child::`a_1 `/.../child::`a_n`]`, i.e., the FXP expression $ch(el(a_1(ch(...ch(el(a_n))))))$. Here we need conjunctive descriptors such as $a_i\&el$ for expressing simultaneous type and tag restrictions for the same node, and negative descriptors such as $\neg el$ and $\neg a_i$ for handling else cases. Thanks to the latter, the size of this eNWA descriptor is in $O(n)$, even though the size of the described eNWA is in $O(n^2)$, since for each of the n states q_i there are n outgoing edges, one for each a_i.

8.5.2 When Variables Must be Bound

Consider the FXP formula $ch(x(true))$. The linearization of an annotated tree must be immediately rejected if x got assigned to the start node, since then x cannot be assigned to any child of the start node anymore. What is relevant here is which variables must be bound in order to make a subformula true.

Definition 11. *Let F be an* FXP *formula and x a variable. We say F must bind x if $F \models fut(x)$ and that F cannot bind x if $F \models \neg fut(x)$.*

In order to approximate $F \models fut(x)$ syntactically, as needed for our automata construction to define rejection states, we define the predicate $F \vdash fut(x)$ as the least binary relation such that:

1. $F_1 \wedge F_2 \vdash \mathit{fut}(x)$ if $F_1 \vdash \mathit{fut}(x)$ or $F_2 \vdash \mathit{fut}(x)$
2. $F_1 \vee F_2 \vdash \mathit{fut}(x)$ if $F_1 \vdash \mathit{fut}(x)$ and $F_2 \vdash \mathit{fut}(x)$
3. $A(F) \vdash \mathit{fut}(x)$ if $F \vdash \mathit{fut}(x)$
4. $L(F) \vdash \mathit{fut}(x)$ if $F \vdash \mathit{fut}(x)$
5. $x(F) \vdash \mathit{fut}(x)$ is true

Given a formula F one can compute in linear time the set $\{x \in \mathcal{V} \mid F \vdash \mathit{fut}(x)\}$. The following soundness lemma for syntactic binding is obvious.

Lemma 8. *For any* Fxp *formula* F *and variable* x*:* $F \vdash \mathit{fut}(x) \Rightarrow F \models \mathit{fut}(x)$.

The converse, i.e. completeness, does not hold in general but still in many interesting cases (see [Lick 2013]). The above soundness result will be sufficient to justify the correctness of our automata construction. In cases where completeness fails our eNwa may fail to be earliest.

8.5.3 Construction of ENWA Descriptors

Let $\mathcal{V}_0 \subseteq \mathcal{V}$ be a finite subset of variables. For any tree t and mapping $\alpha : \mathcal{V}_0 \to \mathrm{nodes}(t)$ we define the annotated tree $t * \alpha$, by replacing in t for any node π the label l of π by $(l, \alpha^{-1}(\pi))$, i.e., by annotating the label of any node by the set of variables that are mapped to it.

Let F be an Fxp formula with variables in $\mathcal{V}_0$ and $n \geq 0$. We define the language of tree suffixes of F with marks at tree depth n as follows:

$$\begin{aligned}\mathcal{L}_n(F) \quad := \quad \{\mathit{suff}(t * \mu, \pi) \mid [\![F]\!]_{t,\pi,\mu} \text{ is true, } \pi \text{ is a node of } t \text{ at depth } n \\ \mu(\mathcal{V}_0) \subseteq \mathit{fut}^t(\pi),\ \mathit{dom}(\mu) = \mathcal{V}_0\}\end{aligned}$$

This language contains all tree suffixes of annotated trees $t*\mu$, such that F evaluates to true for a node π of t at depth n, and variable assignment μ. For any finite set $\mathcal{E}$ of "external" stack symbols, that will be fixed by the context in which F will be used, we construct an eNwa $E_F(\mathcal{E})$, such that for all stacks $S \in \mathcal{E}^n$:

$$\mathcal{L}_S(E_F(\mathcal{E})) = \mathcal{L}_n(F)$$

Note that this equality will hold for all stacks S of height n, so that it is independent of the precise content of the stack.

We call a tree suffix s with opening and closing parentheses `<l>` and `</l>` where $l \in \Sigma \times 2^{\mathcal{V}_0}$ *canonical*, if each variable of $\mathcal{V}_0$ is annotated to exactly one node of s. Whether a tree suffix is canonical can be decided by a streaming algorithm, that runs the following deterministic finite word automaton C on the linearization of s. The state set of C is $2^{\mathcal{V}_0}$, the initial state is $\mathcal{V}_0$, and its final state is $\emptyset$. The rules are $V \xrightarrow{\langle (a,V') \rangle} V \setminus V'$ if $V' \subseteq V$, $V \xrightarrow{\langle /(a,V') \rangle} V$, and $V \xrightarrow{w} V$ for $a \in \Sigma$, $w \in \Delta^*$,

and $V, V' \in 2^{\mathcal{V}_0}$. Note that C gets stuck at the earliest event, when the variable annotation gets in conflict with canonicity.

Given a tree suffix of a marked tree $s = \mathit{suff}(t * \alpha, \pi)$, let $\mathit{Can}(t * \alpha, \pi)$ be obtained from s by annotating all events by the subset of variables in $\mathcal{V}_0$ that were not bound in the past or at the current event by α, so that they can still be bound in the future. More formally $\mathit{Can}(t*\alpha, \pi)$ is the suffix at node π of the tree obtained from $t*\alpha$ by replacing for any node π' the label $(l, V) \in \Sigma \times 2^{\mathcal{V}_0}$ by (l, V, V'), where $V' = \{v \in \mathcal{V}_0 \mid \alpha(v) \in \mathit{fut}^t(\pi') \setminus \{\pi'\}\}$. Note that corresponding opening and closing events of $\mathit{Can}(t * \alpha, \pi)$ have the form $\langle(a, V, V_1)\rangle$ and $\langle/(a, V, V_2)\rangle$ where V_1 may be a proper subset of V_2, so the label of corresponding events need not be the same. Such nested words are still linearizations of trees but with the functions $\mathit{op}(a, V, V_1, V_2)$=`<(a,V,V1)>` and $\mathit{cl}(a, V, V_1, V_2)$=`</(a,V,V2)>`.

First we obtain $\mathit{Can}(s)$ by running the DFA C on the input tree suffix s, whose current state will always be the subset of variables in $\mathcal{V}_0$ that were not yet bound. Second, on the stream $\mathit{Can}(s)$, the automaton $E = E_F(\mathcal{E})$ will run the ENWA described by $D = D_F(\mathcal{E})$ constructed below. Whenever the run of C blocks, the ENWA described by D will go into a rejection state.

8.5.4 Construction of the eNwa descriptor $D_F(\mathcal{E})$

In order to compile an FXP formula to the ENWA descriptor $D_F(\mathcal{E})$, we follow the same fundamental approach as for compiling tree logics such as MSO into tree automata, as used before in the context of XPATH [Gauwin 2009b, Madhusudan 2009]. The most novel part here is the distinction of appropriate selection and rejection states, and the usage of label descriptors. It should also be noticed that our compiler will heavily rely on non-determinism in order to compile formulas with recursive axes such as $\mathit{ch}^+(F)$, $\mathit{ns}^+(F)$, or $\mathit{fo}(F)$, and disjunctions $F_1 \vee F_2$. However, we will try to preserve determinism of the described ENWA as much as possible, so that we can compile many formulas $\neg F$ without having to determinize the described ENWA for F. The compiler will rely on the so-called head $h(D)$, that contains the subset of all opening rules of D that start from an initial state, and the subset of closing rules of D that end in a selection or rejection state with a stack symbol pushed by such an opening rule. Without disjunctions and conjunctions, the heads will always remain of constant size. This is the reason why formulas such as $\mathit{ch}(a(\mathit{ch}(\ldots \mathit{ch}(a)\ldots))$ can be compiled to ENWA descriptors of linear size in linear time. When adding disjunction but no conjunction, the heads are still of amortized constant size, as we will show later on. This is the reason why we will need an amortized size and time analysis.

The construction of $D_F(\mathcal{E})$ is by induction on the structure of F as follows.

Case $F = F_1 \wedge F_2$**.** Let $D_{F_i}(\mathcal{E}) = (A_i, Q_{Si}, Q_{Ri})$ be the ENWA descriptors for F_i where A_i has state set Q_i and $i \in \{1, 2\}$. We define the ENWA descriptor $D = (A, Q_S, Q_R)$ such that A is the product of A_1 and A_2. We choose $Q_S = Q_{S1} \times Q_{S2}$, since a node is safe for selection for $F_1 \wedge F_2$ iff it is safe for selection for both F_1 and F_2. For rejection states we take $Q_R = (Q_{R1} \times Q_2) \cup (Q_1 \times Q_{R2})$, which may lead

to a proper approximation of earliest query answering. Furthermore, note that a large number of conjunctions may lead to an exponential blow-up of the number of states.

Case $F = F_1 \vee F_2$. Let $D_{F_i}(\mathcal{E}) = (A_i, Q_{Si}, Q_{Ri})$ be the ENWA descriptor for the subexpressions where A_i has state set Q_i and $i \in \{1, 2\}$. We define $D = (A, Q_S, Q_R)$ such that A is the union of A_1 and A_2, which introduces nondeterminism in contrast to products. Furthermore, we define $Q_S = Q_{S1} \cup Q_{S2}$ and $Q_R = Q_{R1} \cup Q_{R2}$.

Case $F = \neg F_1$. If $D_1 = D_{F_1}(\mathcal{E})$ describes a deterministic ENWA, then we obtain D by flipping selection and rejection states of D_1. This is correct, since we maintain pseudo-completeness (i.e. no run can ever block [Gauwin 2009a], instead it goes into a rejection state) of the described ENWA as an invariant. There is no approximation here, since a node is safe for selection for $\neg F_1$ iff it is safe for rejection for F_1, and conversely. Otherwise, we first compute the ENWA described by D and determinize it in a first step, which is also free of approximation by Lemma 7, and second apply the previous construction.

Case $F = ch(F_1)$ where F_1 contains neither recursive axes nor disjunctions. Since F_1 neither contains recursive axes nor disjunctions, $D_1 = D_{F_1}(\mathcal{E})$ describes a deterministic ENWA. Furthermore, selection and rejection can be decided no later than when closing the start node. In this case, we can construct D such that it runs D_1 on all children of the start node, deterministically one by one, until a selection state is reached or the start node was closed. This can be done by adding 2 states and stack symbols to D_1 only, based on a recomputation trick for the stack symbol pushed at the start node by D_1 [Gauwin 2009a].

The ENWA described by $D = D(\mathcal{E})$ first reads the opening event of the start node, say $\langle (a, V', V'') \rangle$ and goes into a rejection state if there exists $x \in V'$ such that $F_1 \vdash \mathit{fut}(x)$ or if $V' \cap V'' \neq \emptyset$. In the latter case, C will block. Otherwise, D behaves as D_1 but stacks the Boolean 0 (stating that no previous child was tested successfully) instead of what D_1 would stack. The missing stack symbol will then be recomputed when closing the root of the child. In order to construct D alike, one needs to iterate over the head of D_1, but does not have to touch the rest of D_1. This can be done in time $|h(D_1)|$. The selection states of D are those of D_1, while D introduces a new rejection state. Whenever a run of D_1 goes into a rejection state of D_1 (which is not a rejection state of D), then automaton D may stop any further child tests if x occurs in a set of γ and $F_1 \vdash \mathit{fut}(x)$, in which case D goes into its rejection state. If D reaches the closing event of the start node (that is when the Boolean 0 is popped), then D also goes into the rejection state (this is correct again since F_1 contains no following axes, so that rejection of $ch(F_1)$ can be decided there). Therefore, no external stack symbol from $\mathcal{E}$ will ever be read by D (they may appear only after selection or rejection).

Case $F = A(F_1)$ where $A \in \{ch^+, ns^+, fo\}$ or $F = ch(F_1)$ where F_1 contains recursive axes or disjunctions. Let $D_1 = D_{F_1}(\mathcal{E} \cup \Gamma_{new})$ be the ENWA descriptor for F_1, where Γ_{new} is the set of stack symbols that $D = D_F(\mathcal{E})$ introduces. A nondeterministic ENWA is described by D, which guesses an A-successor of the

start node and runs D_1 starting there. There is a main run of D, which for all non-A successors of the start node goes into a skip state q_{skip}, and which goes into a state q that generates tests for F_1 for potential following A-successors: For $A = ch$ the main run goes to state q such that the next potential opening event is a child of the start node, and it stays in q_{skip} for the subtrees of children of the start node. For $A = ch^+$ the main run goes and stays in state q for all descendants of the start node (no skip state q_{skip} needed). For $A = ns^+$ the main run goes to state q for all siblings to the right of start node, and it stays in q_{skip} for the subtree of the start node and any of the subtrees of its siblings to the right. For $A = fo$ the main run skips the subtree of the start node and after stays in q for the rest of the stream, which is done via closing rules $q \xrightarrow{\langle/(true)\rangle:\beta} q$ for all $\beta \in \mathcal{E}$. The main run of D starts tests for F_1 by adding a rule $q \xrightarrow{\langle a\rangle:\alpha} p$ for each initial rule $i \xrightarrow{\langle a\rangle:\alpha} p$ of D' with $i \in Q_I$ of D'. This main run will continue until:

- either x occurs in a set of γ and $F_1 \vdash \mathit{fut}(x)$, in which case D goes into a rejection state, or
- the closing event of the start node of P arrives (for $A = ch$ and $A = ch^+$),
- the closing event of the parent of the start node of P arrives (for $A = ns^+$),
- the end of the stream arrives (for $A = fo$), indicated by a unique top most stack symbol.

Automaton descriptor D inherits its selection and rejection states from D_1. Note that here it matters again that a candidate can be rejected only if *all* runs of D on this candidate go into a rejection state.

Case $F = B(F_1)$. Without loss of generality, let $B = L_1\& \ldots \& L_m \& x_1 \& \ldots \& x_n$ where $n + m \geq 1$ such that $L_i \in \mathcal{L}$ and $x_i \in \mathcal{V}_0$. Let $D_{F_1}(\mathcal{E})$ be the ENWA descriptor for F_1. We build D from D_1, by restricting all label descriptors of initial rules of D_1 by B: Descriptors $(E_1\& \ldots E_k, (V_1, V_2), (V_1', V_2'))$ are replaced by $(E_1\& \ldots \& E_k \& L_1\& \ldots \& L_m, (V_1 \cup \{x_1, \ldots, x_n\}, V_2), (V_1', V_2'))$. Furthermore, in order to obtain pseudo-completeness, we add one new rule to D for each conjunct of B and each initial state of D_1 going into a rejection state of D_1: For $1 \leq i \leq m$ the new initial rule has label descriptor $(\neg L_i \& L_{i+1} \& \ldots \& L_m, (\emptyset, \emptyset), (\emptyset, \emptyset))$, which describe the complement of $L_1\& \ldots \& L_m$ deterministically. The initial rule for $1 \leq i \leq n$ has label descriptor $(\mathit{true}, (\emptyset, \{x_i\}), (\emptyset, \emptyset))$ describing all variable subsets that do not contain x_i. D inherits selection and rejection states from D_1.

Case $F = O_w$. The ENWA descriptor D for O_w has to compute the concatenation of all strings of text nodes contained in the subtree of the start node, and has to compare it to w with respect to O. This is done by creation of a deterministic finite state automaton B that accepts all strings w' such that (w', w) is in the relation induced by O. The idea is that D runs automaton B on all strings of text nodes in the subtree of the start node. D maintains a copy of states of B, called *copy states*. D's main run r remains in copy states of B whenever not reading strings of text

nodes of the subtree of the start node, while r switches to corresponding states of B at text nodes. The main run starts in the initial copy state of B and stays there until the first text node arrives at which it changes to the initial state of B. The string of the text node is consumed by B, while reaching some state q which is not necessarily a final state of B. At the corresponding closing event of the text node, the main run goes into the corresponding copy state q_{copy} for q and stays there until the next text node arrives, where r continues as before. The main run continues until the closing event of the start node, where O_w can be decided. Whenever B blocks, the main run moves into a rejection state, and whenever a final state of B is reached, it moves into a selection state.

Case $F = true$. The ENWA descriptor D for *true* has five plus $2|\mathcal{E}|$ rules with label descriptor $(true, (\emptyset, \emptyset), (\emptyset, \emptyset))$: one opening initial rule to a selection state, opening and closing rules looping in the selection and the rejection state, and $2|\mathcal{E}|$ closing rules to close the stream in the selection and rejection state.

Definition 12. *We call an* FXP *formula* determinization free *if it does not contain any disjunction or any axis from ch^+, ns^+, and fo below a negation.*

For the complexity analysis of our compiler, we will need to assume that the input FXP formula is *simple*, in that all label subformulas are of one of the following three forms: $B(O_w)$, $B(A(F))$ or $B(true)$. We can transform all FXP formulas into equivalent simple FXP formulas by applying the rewrite rule $B(F) \rightarrow B(true) \wedge F$ everywhere. This procedure, however, would increase the conjunction width, on which the efficiency of our compiler depends. Instead, we will assume that the input formulas is simply labeled in the following sense.

Definition 13. *We call an occurrence of an operator in a formula* labeled *if it has some ancestor, which is a label formula B but without having any axis A in between. We call a formula* simply labeled *if it does not contain any labeled disjunctions and negations.*

Lemma 9. *Any simply labeled* FXP *formula can be transformed into a simple* FXP *formula with the same conjunction width in linear time.*

Proof. We apply the following two rewrite rules exhaustively in a bottom-up manner. This can be done in linear time since no subformulas are copied, and it does not affect the number of conjunctions.

$$\begin{array}{lcl} B(F \wedge F') & \rightarrow & B(F) \wedge F' \\ B(B'(F)) & \rightarrow & B\&B'(F) \end{array}$$

Both rules preserve simple labeling, since whenever one of them applies, all unlabeled disjunctions and negations of F and F' must be located below some axis. Therefore, they will remain unlabeled. If none of these rules applies any more, the resulting FXP formula is simple: subformulas $B(F \wedge F')$ and $B(B'(F))$ would be rewritten, $B(F \vee F')$ and $B(\neg(F))$ are excluded since not simply labeled, and all other three possible forms $B(true)$, $B(A(F))$ and $B(O_w)$ are simple. □

8.5.5 Correctness

Proposition 8 (Correctness)**.** *Let $\mathcal{E}$ be a set of "external" stack symbols, $n \geq 0$, $S \in \mathcal{E}^n$, F an* FXP *formula with variables in a finite set $V \subseteq \mathcal{V}_0$, and $D = D_F(\mathcal{E})$ be the constructed* eNWA*. Then*

$$\mathcal{L}_S(D_F(\mathcal{E})) = Can(\mathcal{L}_n(F))$$

Proof. The proof is by induction on the structure of FXP formulas.

Case $F = F_1 \wedge F_2$. Let $D_1 = D_{F_1}(\mathcal{E})$ and $D_2 = D_{F_2}(\mathcal{E})$ be the eNWA for F_1 and F_2 respectively. Let $s \in Can(\mathcal{L}_n(F))$, then trivially, $s \in Can(\mathcal{L}_n(F_1))$ and $s \in Can(\mathcal{L}_n(F_2))$. By induction, for any $i \in \{1, 2\}$, some run on s by D_i goes into a selection state. Selection states of D are pairs of a selection state of D_1 and a selection state of D_2. Hence some run of D on s goes into a selection state as well, such that $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then there exist some run on s by D which goes into a selection state. Therefore there exist some run on s by D_i going into a selection state of D_i. By induction it follows that $s \in Can(\mathcal{L}_n(F_i))$ for $i = \{1, 2\}$, and thus $s \in Can(\mathcal{L}_n(F))$.

Case $F = F_1 \vee F_2$. Let $D_1 = D_{F_1}(\mathcal{E})$ and $D_2 = D_{F_2}(\mathcal{E})$ be the eNWA for F_1 and F_2 respectively. Let $s \in Can(\mathcal{L}_n(F))$, then there exist $i \in \{1, 2\}$ such that $s \in Can(\mathcal{L}_n(F_i))$. By induction, some run on s by D_i goes into a selection state. Since all selection states of D_i are selection states of D it follows that some run of D on s goes into a selection state, and $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then there exist some run on s by D which goes into a selection state. Hence there exist $i \in \{1, 2\}$ such that some run on s by D_i goes into a selection state. By induction it follows that $s \in Can(\mathcal{L}_n(F_i))$, and hence $s \in Can(\mathcal{L}_n(F))$.

Case $F = \neg F_1$. Let $D_1 = D_{F_1}(\mathcal{E})$ be the eNWA for F_1. Let $s \in Can(\mathcal{L}_n(F))$, then $s \notin Can(\mathcal{L}_n(F_1))$. By induction, all runs on s by D_1 go into a rejection state. By Lemma 7 the unique run of the determinization of D_1 on s goes into a rejection state. Since D flips selection with rejection states the unique run of D on s goes into a selection state, and thus $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then the unique run on s by D goes into a selection state. Hence the unique run by D_1 on s goes into a rejection state. By induction it follows that $s \notin Can(\mathcal{L}_n(F_1))$, so that $s \in Can(\mathcal{L}_n(F))$.

Case $F = ch(F_1)$ where F_1 contains neither recursive axes nor disjunctions. Let $D_1 = D_{F_1}(\mathcal{E})$ be the eNWA for F_1. Let $s \in Can(\mathcal{L}_n(F))$, then there exist a suffix s' of s with $s' \in Can(\mathcal{L}_n(F_1))$, while starting with the opening event of some child of the start node of s. By induction, as D_1 is deterministic the unique run on s' by D_1 goes into a selection state. By construction the unique run of D goes into the same selection state on s. Hence $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then the unique run on s by D goes into a selection state. As D runs D_1 on all children of the start node of s, and as D goes into a selection state whenever D_1 does, it follows that the unique run of D_1 goes into a selection state for some suffix s' of s, that starts with the opening event of some child of the start node of s. By induction it follows that $s' \in Can(\mathcal{L}_n(F_1))$, so that $s \in Can(\mathcal{L}_n(F))$.

Case $F = A(F_1)$ where $A \in \{ch^+, ns^+, fo\}$ or $F = ch(F_1)$ where F_1 contains recursive axes or disjunctions. Let $D_1 = D_{F_1}(\mathcal{E} \cup \Gamma_{new})$ be the ENWA for F_1, where Γ_{new} are stack symbols that D introduces. Let $s \in Can(\mathcal{L}_n(F))$, then there exist a suffix s' of s with $s' \in Can(\mathcal{L}_n(F_1))$, while starting with the opening event of some A-successor of the start node of s. By induction, some run of D_1 on s' goes into a selection state. By construction some run of D goes into the same selection state on s. Thus $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then some run on s by D goes into a selection state. As D runs D_1 on all A-successor of the start node of s, and as D goes into a selection state whenever D_1 does, it follows that some run of D_1 goes into a selection state for some suffix s' of s, that starts with the opening event of some A-successor of the start node of s. By induction it follows that $s' \in Can(\mathcal{L}_n(F_1))$, and hence $s \in Can(\mathcal{L}_n(F))$.

Case $F = B(F_1)$. Let $D_1 = D_{F_1}(\mathcal{E})$ be the ENWA for F_1. Let $s \in Can(\mathcal{L}_n(F))$, then the label of the start node of s belongs to all label properties L in B, while the start node is annotated according to all variable restrictions in B, and $s \in Can(\mathcal{L}_n(F_1))$. By induction, some run r of D_1 on s goes into a selection state. By construction as D runs D_1 for some start node, whose label belongs to all label properties L of B and is annotated by all variable restrictions in B, the same run r of D goes into the same selection state on s. Therefore $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$, then some run r on s by D goes into a selection state only if the label of the start node satisfies all properties or restrictions in B. This selection state is a selection state of D_1, such that the same run r of D_1 goes into a selection state on s. By induction it follows that $s \in Can(\mathcal{L}_n(F_1))$, and therefore $s \in Can(\mathcal{L}_n(F))$ according to fulfilled label and annotation restrictions.

Case $F = O_w$. Let $s \in Can(\mathcal{L}_n(F, V))$, then the start node of s belongs to O_w. By construction D runs a finite state automaton B (which accepts all string w', such that (w, w') is in the relation induced by O) over the concatenation of the text nodes in the subtree of the start node of s. As the start node of s belongs to O_w, D goes into a final state of B for some string of some text node, and thereby into a selection state of D. Hence $s \in \mathcal{L}_S(D)$. Conversely let $s \in \mathcal{L}_S(D)$. Then there is a run on s by D which goes into a selection state, such that the data value of the start node of s must be in relation with w with respect to O. Hence the start node of s belongs to O_w, such that $s \in Can(\mathcal{L}_n(F, V))$.

Case $F = true$. $s \in Can(\mathcal{L}_n(F))$, is true for any tree suffix s. By construction D moves to a selection state for any start node of s. Hence any run of D on any suffix s goes into a selection state, such that $s \in \mathcal{L}_S(D)$.

□

8.5.6 Size of Automata Descriptors

Lemma 10 (Size of automaton descriptor)**.** *There exists a constant c' such that for any* FXP*-formula F, that is determinization free and simply labeled, and for any finite set $\mathcal{E}$ of external stack symbols: $|D_F(\mathcal{E})| \leq (|F|(c' + 2|\mathcal{E}|))^{w(F)+1}$.*

Proof. By Lemma 9 we can make F simple in a preprocessing step in linear time without changing the conjunction width $w(F)$. For the treatment of recursive axis ch^+, ns^+, and fo, we need nondeterministic automata with multiple initial rules. If we always had a single initial rule, then the proof of the lemma was straightforward. With multiple initial rules, we need an amortized cost analysis. Therefore, we define the amortized size of an eNWA descriptor D by $AS(D) = |D| + |init(D)|$ where $|D|$ is the size of D and $init(D)$ the set of initial rules of D, i.e. rules that depart an initial state. With the constant $c' = 7$, we can show for all determinization free FXP formulas F that:

$$AS(D_F(\mathcal{E})) \leq (|F|(c' + 2|\mathcal{E}|))^{w(F)+1}$$

The lemma will then follow from $|D_F(\mathcal{E})| \leq AS(D_F(\mathcal{E}))$. The proof is by induction on the structure of F. Let $D = D_F(\mathcal{E})$. Let $\mathcal{E}'$ be the extension of $\mathcal{E}$ and F_1 and possibly F_2 the subformulas of F, such that D was constructed from $D_1 = D_{F_1}(\mathcal{E}')$ and possibly $D_2 = D_{F_2}(\mathcal{E}')$. We write ω, ω_1, and ω_2 for the respective conjunction widths of $w(F)$, $w(F_1)$, and $w(F_2)$, and I, I_1 and I_2 for the number of initial rules of D, D_1, and respectively D_2.

Case $F = F_1 \wedge F_2$. The eNWA descriptor D is the product D_1 and D_2 where $\mathcal{E} = \mathcal{E}'$. Hence $|D| = |D_1|\ |D_2|$ and $I = I_1\ I_2$, so that:

$$\begin{array}{rcll} AS(D) & \leq & AS(D_1)\ AS(D_2) & \\ & \leq & (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1}\ (|F_2|(c' + 2|\mathcal{E}|))^{\omega_2+1} & \text{(ind. hypo.)} \\ & \leq & (|F|(c' + 2|\mathcal{E}|))^{\omega_1+1} \cdot (|F|(c' + 2|\mathcal{E}|))^{\omega_2+1} & \\ & = & (|F|(c' + 2|\mathcal{E}|))^{\omega_1+\omega_2+2} & \\ & = & (|F|(c' + 2|\mathcal{E}|))^{\omega+1} & \end{array}$$

Case $F = F_1 \vee F_2$. The eNWA descriptor D is the union of the eNWA descriptors D_1 and D_2 where $|\mathcal{E}| = |\mathcal{E}|'$, so $|D| = |D_1| + |D_2|$ and $I = I_1 + I_2$. It follows that $AS(D) = AS(D_1) + AS(D_2)$ and thus smaller than in the case of conjunction.

Case $F = \neg F_1$. Since F is determinization free, D_1 describes a deterministic automaton, which furthermore can never get stuck, so that we only need to swap rejection and selection states of D_1 in order to obtain D. Hence, by induction hypothesis: $AS(D) = AS(D_1) \leq (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$

Case $F = ch(F_1)$ where F_1 contains neither recursive axes nor disjunctions. Since F_1 contains no nondeterministic constructs, the automaton described by D_1 will be deterministic. The eNWA described by D will thus run the eNWA described by D_1 (where $\mathcal{E} = \mathcal{E}'$) on all children of the marked node, until one of these runs succeeds. The rules of the head of D_1 are rewritten for recomputing the stack symbols that D_1 used at the roots of all children of the marked node, but the number of rules is not increased thereby. In addition, $c' = 7$ new rules were added for the treatment of the marked node. Therefore, $|D| = |D_1| + c'$. The size of the head of D is smaller than that of D_1. Hence, $AS(D) \leq c' + AS(D_1) \leq c' + (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = A(F_1)$ where $A \in \{ch^+, ns^+, fo\}$ or $F = ch(F_1)$ where F_1 contains recursive axes or disjunctions. Now $\mathcal{E}'$ is obtained by $\mathcal{E}$ by adding at most c' new stack symbols. The eNwa descriptor D has a generator state, from which it starts all initial rules I_1 of D_1, when being at an A-successor of the marked node. For $A = fo$ maximum $|\mathcal{E}|$ rules are added in order to find all following nodes in any future. In addition, $c' = 7$ new rules were added for the treatment of the main run of D. Thus, $|D| \leq |D_1| + I_1 + c' + |\mathcal{E}|$, so that:

$$\begin{aligned} AS(D) &= |D_1| + I_1 + c' + |\mathcal{E}| \\ &= AS(D_1) + c' + |\mathcal{E}| \\ &\leq (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + c' + |\mathcal{E}| \quad \text{(ind. hypo.)} \\ &\leq ((|F_1| + 1)(c' + 2|\mathcal{E}|))^{\omega_1+1} \\ &= (|F|(c' + 2|\mathcal{E}|))^{\omega+1} \end{aligned}$$

Case $F = B(F_1)$. Since F is simply labeled, F_1 has the form $A(F_2)$ or O_w or *true*. Hence, descriptor D_1 for F_1 has only one initial state. The size $|D|$ of the eNwa descriptor D is the size of D_1, plus $|B|$ opening rules to the initial state of D_1, with which the complement of B is described (for pseudo-completeness). Hence $|D| = |D_1| + |B|$ and the number of initial rules I of D is $I_1 + |B|$. Therefore $AS(D) = |D_1| + |B| + I = AS(D_1) - I_1 + |B| + I_1 + |B| \leq (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + 2|B| \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$, since $2 \leq c'$ and $|F| = |F_1| + |B|$.

Case $F = O_w$. The size of D is the sum of at most $3 \cdot |w|$ many rules for the finite state automaton B (that accepts all strings w' such that (w', w) is in the relation induced by O), $3 \cdot |w|$ many rules for moving in and out of copy states of B as described above, and at most c' many rules for the treatment of Ds main run. In addition D needs $2|\mathcal{E}|$ rules to close the stream in selection and rejection states. Hence the size of D is smaller or equal to $6|w| + c' + 2|\mathcal{E}|$, while the head of D is constant and can be bounded by c'. As $|F| = |w|$ it follows that $AS(D) \leq (|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = true$. The size $|D|$ of the eNwa descriptor D for *true* is 5 and contains $2|\mathcal{E}|$ many rules to close the stream. D's amortized size is thereby smaller than $(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

□

8.5.7 Time to Compute Automata Descriptors

Theorem 2 (Compilation time)**.** *There exist $c, c' > 0$ such that for any determinization free and simply labeled* Fxp*-formula F and any finite set $\mathcal{E}$ of external stack symbols, the time to compute the* eNwa *descriptor $D_F(\mathcal{E})$ is at most $2c(|F|(c' + 2|\mathcal{E}|))^{w(F)+1}$.*

Proof. We fix a set $\mathcal{E}$ and constants $c_1 = 2$, $c' = 7$ and $c = c_1 + c' = 9$. We again need an amortized analysis. We define the amortized time of an eNwa descriptor $D = D_F(\mathcal{E})$ constructed by our algorithm as follows, where $T(D)$ is the construction

time:

$$AT(D) = T(D) + |h(D)| + |\mathcal{E}|$$

The proof is by induction on the structure of FXP formulas F, under the assumption that they are determinization free and simply labeled. Let F_1 and possibly F_2 be the arguments of the top-level operator of F, and $D_i = D_{F_i}(\mathcal{E}')$ be the subautomata from which D was constructed. We will also write H for $|h(D)|$, H_i for $|h(D_i)|$, and similarly T for $T(D)$, T_i for $T(D_i)$.

Case $F = F_1 \wedge F_2$. The time to compute an ENWA descriptor $D = D_F(\mathcal{E})$ for the product F is the sum of the times for computing the ENWA descriptors $D_1 = D_{F_1}(\mathcal{E})$ and $D_2 = D_{F_2}(\mathcal{E})$, plus the time to compute ENWA descriptor D for F, which can be done in time $c_1 \cdot |D_1| \cdot |D_2|$. Hence

$$\begin{array}{rcll}
AT(D) & = & T_1 + T_2 + c_1\ |D_1|\ |D_2| + H + |\mathcal{E}| & \\
& \leq & AT(D_1) + AT(D_2) + c_1\ |D_1|\ |D_2| + H & \\
& \leq & AT(D_1) + AT(D_2) + (c_1 + 1)\ |D_1|\ |D_2| & \\
& \leq & 2c\ (|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + 2c\ (|F_2|(c' + 2|\mathcal{E}|))^{\omega_2+1} & \\
& & +(c_1 + 1)\ |D_1|\ |D_2| & \\
& \leq & 2c\ ((|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + (|F_2|(c' + 2|\mathcal{E}|))^{\omega_2+1} & (c_1 + 1 \leq c) \\
& & +|D_1|\ |D_2|) & \\
& \leq & 2c\ ((|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + (|F_2|(c' + 2|\mathcal{E}|))^{\omega_2+1} & (\textit{Lemma 10}) \\
& & +(|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1})\ (|F_2|(c' + 2|\mathcal{E}|))^{\omega_2+1} & \\
& \leq & 2c\ ((|F_1| + |F_2|)(c' + 2|\mathcal{E}|))^{\omega_1+\omega_2+2} & (a^m + b^n + a^m b^n \leq (a+b)^{m+n}) \\
& \leq & 2c\ (|F|(c' + 2|\mathcal{E}|))^{\omega+1} &
\end{array}$$

Case $F = F_1 \vee F_2$. The time to compute an ENWA descriptor D is the sum of the times to compute D_1 and D_2. The size of the head of D is the sum of the head of D_1 and the head of D_2. Hence

$$\begin{array}{rcl}
AT(D) & = & T + H + |\mathcal{E}| = T_1 + T_2 + H + |\mathcal{E}| = \\
& = & AT(D_1) - H_1 - |\mathcal{E}| + AT(D_2) - H_2 - |\mathcal{E}| + H + |\mathcal{E}| \\
& \leq & AT(D_1) + AT(D_2)
\end{array}$$

and thus smaller than in the case of conjunction.

Case $F = \neg F_1$. Since F is determinization free, D_1 describes a deterministic automaton, which furthermore can never get stuck, so that we only need to swap rejection and selection states of D_1 in order to obtain D. Hence, by induction hypothesis: $AT(D) = AT(D_1) \leq 2c(|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} \leq 2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$

Case $F = ch(F_1)$ where F_1 contains neither recursive axes nor disjunctions. Since F_1 contains no nondeterministic constructs, the automaton described by D_1 will be deterministic. The time to compute D is the time to compute D_1 (where $\mathcal{E} = \mathcal{E}'$), plus the time to rewrite the head of D_1 for the recomputation of stack symbols pushed at root of the children of the marked node. In addition D takes time $2|\mathcal{E}|$ to delete closing stream rules for the rejection state of D_1 and to add these to the new rejection state of D_1, and time at most c' to add at most c' new

rules for the treatment of the marked node. The head of D contains only 5 rules. Hence

$$\begin{array}{rcll} AT(D) & = & AT(D_1) - H_1 - |\mathcal{E}| + H_1 + 2|\mathcal{E}| + c' + H + |\mathcal{E}| & \\ & = & AT(D_1) + 2|\mathcal{E}| + c' + 5 & \\ & \leq & 2c(|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + 2|\mathcal{E}| + c' + 5 & \text{(ind. hypo.)} \\ & \leq & 2c((|F_1| + 1)(c' + 2|\mathcal{E}|))^{\omega_1+1} & 5 \leq c \\ & = & 2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1} & \end{array}$$

Case $F = A(F_1)$ where $A \in \{ch^+, ns^+, fo\}$ or $F = ch(F_1)$ where F_1 contains recursive axes or disjunctions. The time to compute D is the time to compute D_1 where $\mathcal{E}'$ is obtained from $\mathcal{E}$ by adding at most c' new stack symbols. Furthermore D needs time to compute I_1 many rules for starting the test for F_1 from Ds generator state, plus $|\mathcal{E}|$ closing rules to reach all following nodes of the stream for the case that $A = fo$, plus at most c' rules for the treatment of the main run. The head $H = H_1 - I_1 + 2$, as the head of D contains all closing rules of the head of D_1, but none of the initial rules belonging to the head of D_1, such that D needs two new initial rules. Therefore $AT(D) = T(D) + h(D) + |\mathcal{E}| =$

$$\begin{array}{rcl} AT(D) & = & T_1 + I_1 + |\mathcal{E}| + c' + h(D) + |\mathcal{E}| \\ & = & AT(D_1) - H_1 - |\mathcal{E}| + I_1 + |\mathcal{E}| + c' + H_1 - I_1 + 2 + |\mathcal{E}| \\ & = & AT(D_1) + c' + 2 + |\mathcal{E}| \end{array}$$

and thus smaller than in the case of $F = ch(F_1)$ where F_1 contains no nondeterministic constructs.

Case $F = B(F_1)$. Since we assumed that F is simply labeled, it follows that F_1 is either a comparison O_w, an axis $A(F_2)$, or *true*. In all cases, the eNWA descriptor D_1 for F_1 has only one initial state, and its head H_1 is of size at most c'. Therefore the time T for computing D is the time to compute D_1, plus the time to intersect label properties and variable restriction in B with the label descriptors of initial rules of D_1 (of which there are at most H_1 many), plus the time to add an opening rule for the initial state of D_1 to some rejection state of D_1 to describe the complement of B. The head of D contains all rules from the head of D_1, plus $|B|$ rules for the initial state in order to maintain pseudo-completeness. Hence

$$\begin{array}{rcll} AT(D) & = & T + H + |\mathcal{E}| = T_1 + c' + |B| + H + |\mathcal{E}| & \\ & = & AT(D_1) - H_1 - |\mathcal{E}| + c' + |B| + H_1 + |B| + |\mathcal{E}| & \\ & = & AT(D_1) + c' + 2|B| & \\ & \leq & 2c(|F_1|(c' + 2|\mathcal{E}|))^{\omega_1+1} + c' + 2|B| & \text{(ind. hypo.)} \\ & \leq & 2c(|F|(c' + 2|\mathcal{E}|))^{\omega_1+1} & |F| = |F_1| + |B| \end{array}$$

Case $F = O_w$. The time to compute D is the time to compute at most $3 \cdot |w|$ many rules for the finite state automaton B (that accepts all strings w' such that (w', w) is in the relation induced by O), the time to compute $3 \cdot |w|$ many rules for moving in and out of copy states of B as described above, the time to compute at most c'

many rules for the treatment of Ds main run, plus $2|\mathcal{E}|$ rules to close the stream in selection and rejection states. D's head contains only four rules. With $|F| = |w|$ the amortized time to compute D is therefore at most $6|w| + c' + 2|\mathcal{E}| + 4 + |\mathcal{E}| \leq 2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

Case $F = \mathit{true}$**.** The time to compute D is the time to compute the $5 + 2|\mathcal{E}|$ many rules. The head of D has size 3 (initial rule plus 2 closing rules in for the selection and rejection state). D's amortized time is therefore $5 + 2|\mathcal{E}| + 3 + |\mathcal{E}| = 8 + 3|\mathcal{E}|$, which is smaller than $2c(|F|(c' + 2|\mathcal{E}|))^{\omega+1}$.

□

8.6 Early Query Answering

We show how to use eNWA descriptors for evaluating monadic queries on XML streams. The main idea is to generate on the fly all possible answer candidates, and to run the described eNWA on all of them in parallel in a streaming manner. The nondeterminism and the automata descriptors are resolved by on-the-fly instantiation and determinization. We then improve this streaming algorithm in an important manner, so that the stacks and states of the runs of multiple answer candidates in the same state may be shared.

8.6.1 On-the-fly Instantiation and Determinization

Let D be a descriptor of an eNWA E' that defines a monadic query, i.e., with tag alphabet $\{a, a^x \mid a \in \Sigma\}$ where x is a fixed variable. We are interested in running the determinization E of E'. This can be done while generating the needed part of E from D on the fly (and thus without ever constructing E'). At any time point, we store the subset of the states and transitions of E that were used before. If a missing transition rule is needed for some state $\{(q_1, q'_1), \ldots, (q_k, q'_k)\}$ of E and some label l then for each $1 \leq i \leq k$ one computes transitions of D from state q_i that describe label l (if not already computed once). These transition rules are then used to compute the result state of E by applying the determinization procedure. It should be noticed that each transition of E can be computed in polynomial time (but not in linear time). For example, to compute a missing closing transition of the deterministic automaton by on-the-fly determinization, our algorithm needs time $O(|Q|^4 \cdot |\Sigma|)$, where Q are the states of E', see Section 2.4.4. Recall also that the states of E are sets of pairs of states of E'. For efficiency reasons, we will substitute such sets by integers, so that the known transitions of E can be executed as efficiently as if a deterministic eNWA E' was given at beforehand. Therefore, we can safely separate the aspects of on-the-fly instantiation and determinization from what follows.

8.6.2 Streaming Algorithm for Deterministic ENWAs

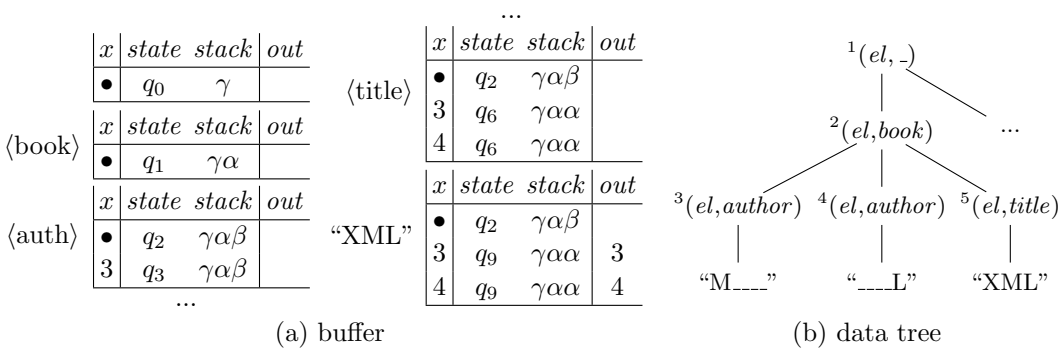
We present a streaming algorithm that answers monadic queries defined by a deterministic eNWAs E in an early manner. This algorithm can then be lifted to eNWA descriptors by on-the-fly determinization and instantiation, as explained above. We will also assume that the deterministic eNWA E is pseudo-complete, so that it has a unique complete run on any document.

Buffering Possibly Alive Candidates Suppose that we are given a stream containing a nested word of some data tree, and that we want to compute the answers of the query defined by E on this data tree in an early manner. That is, we have to find all nodes of the data tree that can be annotated by x, so that E can run successfully on the annotated data tree. At any event e of the stream, our algorithm maintains a finite set of candidates in a so-called buffer. A candidate is a triple that contains a value for x, a state of E that is neither a selection nor a rejection state, and a stack of E. The value of x can either be a node opened before the current event e, or "unknown" which we denote by $\bullet$. At the start event, there exists only a single candidate in the buffer, which is $(\bullet, q_0, \bot)$ where q_0 is the unique initial state of E and $\bot$ the empty stack. At any later event, there will be at most one candidate containing the $\bullet$.

Lazy Candidate Generation New candidates are generated lazily under supervision of the automaton. This can happen at all opening events for which there exists a candidate with the unknown value $\bullet$ (which then is unique). Consider the `<a>` event of some node π and let $(\bullet, q, S)$ be the candidate with the unknown value in the buffer at this event. The algorithm then computes the unique pair (γ, q') such that $q \xrightarrow{\langle a^x \rangle : \gamma} q'$ is a transition rule of E. If q' is a selection state, then π is an answer of the query, so we can output π directly. If q' is a rejection state, then π is safe for rejection (since E is deterministic), so we can ignore it. Otherwise, π may still be alive, so we add the candidate $(\pi, q', S\gamma)$ to the buffer.

Candidate Updates At every event, all candidates in the buffer must be updated except for those that were newly created. First, the algorithm updates the configuration of the candidate by applying the rule of E with the letter of the current event to the configuration of the candidate. If a selection state is reached, the node of the candidate is output and discarded from the buffer. If a rejection state is reached, the candidate is also discarded from the buffer. Otherwise, the node may still be alive, so the candidate is kept in the buffer.

Example We illustrate the basic algorithm in Figure 8.8 on the eNWA from Figure 8.3 and the suffix of the document from Figure 8.2 with the book as start node. Initially the buffer contains a single candidate with the unknown node $\bullet$, that starts in the initial state of the eNWA. According to opening tag $\langle \mathit{library} \rangle$ we launch the open transition and apply state and stack changes. At the opening event of node



(a) buffer (b) data tree

Figure 8.8: Evolution of the buffer (a) for the eNwa from Fig. 8.3 when answering the XPath query `book[starts-with(title,'XML')]/auth` on the suffix of the data tree (b) with start node book.

3, i.e., when reading the open tag $\langle author \rangle$ in state q_1, a new candidate is created. This is possible, since there exists the transition rule $q_1 \xrightarrow{\langle author^x \rangle : \beta} q_3$ in the eNwa and since q_3 is neither a rejection nor a selection state. Similarly a new candidate will be created for node 4 at its opening event. Only after having consumed the text value of the title node 5, a selection state is reached for the candidates with node 3 and 4, such that they can be output and removed from the buffer.

8.6.3 Adding Stack-and-State Sharing

For most queries of the XPathMark benchmark, the buffer will contain only 2 candidates at every event, of which one is the candidate with the unknown value •. It may happen though that the number of candidates grows linearly with the size of the document. An example is the XPath query `/child::a[following::b]` on a document whose root has a large list of only a-children. There the processing time will grow quadratically in the size of the document. All candidates (of which there are $O(n)$ for documents of size n) must be touched for all following events on the stream (also $O(n)$). A quadratic processing time is unfeasible even for small documents of some megabytes, so this is a serious limitation.

We next propose a data structure for state and stack sharing, that allows to solve this issue. The idea is to share the work for all candidates in the same state, by letting their stacks evolve in common. Thereby the processing time per event for running the eNwa on all candidates will become linear in the number of states and stack symbols of the eNwa, instead of linear in the number of candidates in the buffer. In addition to this time per event, the algorithm must touch each candidate at most three times, once for creation, output, and deletion. We will use a directed acyclic graph (Dag) with nodes labeled in Γ for sharing multiple stacks. For instance a Dag for stacks $\gamma_1\gamma_2\gamma_3\gamma_4$ and $\gamma_1\gamma_2'\gamma_3\gamma_4'$ share nodes for stack symbols γ_1 and γ_3, as depicted in Figure 8.9.

In addition, we use a table $B : Q \times \Gamma \to \mathit{Aggreg}$ relating a state and a root of

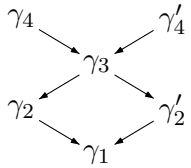


Figure 8.9: DAG representing stacks.

the DAG through an aggregation of candidates. Table B aims at storing enough information when sharing at opening events, so that one can undo the sharing properly at closing events. Formally, *Aggreg* is the least set such that (1) it contains all sets of candidates, and (2) all subsets of $\{(a, \gamma) \mid a \in \mathit{Aggreg}, \gamma \in \Gamma\}$. For instance, the DAG and its associated B table at the `<title>`-event in Figure 8.8 is illustrated in Figure 8.10. Here we have $B(q_6, \alpha) = \{3, 4\}$ and $B(q_2, \beta) = \{\bullet\}$. In this case, the aggregations are just sets of candidate nodes. In general, an aggregation contains information on how to unshare candidates related to a root in the DAG which represents more than one stack. This means one needs more information than just a set of candidates, and for this reason we introduce a nested structure. Let $B(q_i, \beta_i) = A_i$ for aggregations A_i in state q_i with stack symbol β_i on top of the stack, for $1 \leq i \leq k$ (i.e. there are k roots).

At an opening event the automaton may have transitions that lead into the same state q pushing the same stack symbol β for some of the states q_i, for instance q_2 and q_3. Since the corresponding aggregations A_2 and A_3 originate from different roots in the DAG, we add this information in a new aggregation $A = \{(A_2, \beta_2), (A_3, \beta_3)\}$. Consider for instance the situation depicted in Figure 8.11. From the first configuration, we reach the second with the `<a>`-event. There the candidate 2 is created from the $\bullet$-candidate whose configuration has β on top of the stack, goes into state q'', and pushes γ'. However, there is also the candidate 1 which will go into the same state q'' while pushing the same stack symbol γ', but from a configuration with β' on top of the stack. The pairs $(2, \beta)$ and $(1, \beta')$ must be stored in the aggregation, so we define $B(q'', \gamma') = \{(2, \beta), (1, \beta')\}$.

At closing events for aggregations that contain pairs (A_i, β_i) of an aggregation A_i and a stack symbol β_i, the unsharing is done by relating aggregation A_i to the new root β_i in the DAG, after closing. For instance the closing event `</a>` is read when moving from the second to the third situation in

Figure 8.11. There we have to undo the sharing. We decompose the aggregate and update the data structure to $B(q''', \beta) = \{2\}$, $B(q''', \beta') = \{1\}$ and $B(q, \beta) = \{\bullet\}$.

At every time point, B contains each buffered candidate exactly once. Whenever a selection state is reached in the B-table, the candidates in the aggregate of this state will be output and the aggregate will be deleted from the data structure. For rejection states, we only have to discard the aggregate. Note that rejected or selected candidates get deleted entirely from the data structure this way, since no candidate may appear twice in different aggregates, again due to determinism.

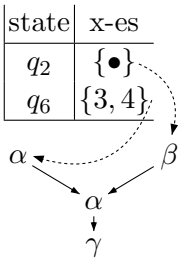


Figure 8.10: Buffer of `<title>`.

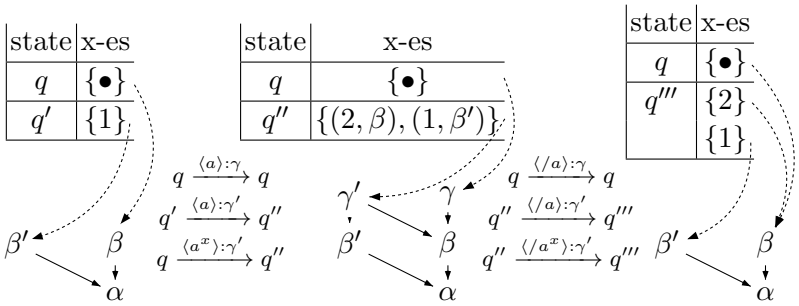


Figure 8.11: Data structures for the state sharing algorithm.

Proposition 9 (Time per Event)**.** *For every deterministic* eNwa E *defining a monadic query* P *and data tree* t *the time complexity per event of our query answering algorithm with stack-and-state sharing to compute* $P(t)$ *is in* $\mathcal{O}(|Q|)$*, where* Q *is the set of states of* E*.*

We can even reduce $|Q|$ to the maximal number of states per event that are assigned to the buffered candidates. This bound is at most 2 for all queries in our practical experiments. In practice, therefore, the runtime should grow linearly in the size of the document, and be independent of the size of the query. This will indeed be confirmed by the experiments in Figure 8.18, which also indicate that the costs for on-the-fly instantiation and determinization are irrelevant in practice.

Proof. We start with the case of opening events. Since E is deterministic there is only one opening transition per state and label. According to the tag alphabet $\{a, a^x \mid a \in \Sigma\}$, the query answering algorithm has therefore two choices for opening transitions for an opening event a. Hence at most $2\,|Q|$ edges to new roots in the Dag may be added. For candidates in table B additional information may have to be stored (which needs only constant time). This happens, when candidates in different states related to different roots in the Dag reach the same state at an opening event. The evolution of the Dag at opening events can therefore be done in time $\mathcal{O}(|Q|)$.

For internal events, the Dag structure is not altered. At most $|Q|$ many state changes occur in table B.

We finish with the case of closing events. Closing transitions are applied according to the label of the closing event e' and to table B, which relates states and roots of the DAG to an aggregation of candidates. At first glance it seems as there could be $|Q|\ |\Gamma|$ (hence quadratically many) possible transitions. Nevertheless the number of possible closing transitions at e' is bounded by

1. the number of opening transitions for opening event e that corresponds to e' (of which there are maximum $2\ |Q|$ many), plus

2. those closing transitions for potential candidates that were created between events e and e'.

(1) is obvious, in that at event e table B contained maximum $2\ |Q|$ entries to aggregations C of candidates. As E is deterministic at event e' there are thus $2\ |Q|$ many possibilities for these aggregations C to be related with stack symbols on top of the DAG. For (2) let us consider the creation of new candidates between events e and e'. All these candidates were instantiated from the unique $\bullet$ candidate, such that they share parts of the stack. At the closing event all potentially created candidates therefore are related to the unique stack symbol γ in the top of the DAG belonging to the $\bullet$ candidate. These new candidates may be in at most $|Q|$ many different states. Hence with (1) and (2) maximum $3\ |Q|$ closing transition may be applied. For these only a constant number of operations are needed to unshare the aggregations of candidates. Hence all together the run-time per event is in $\mathcal{O}(|Q|)$. □

Theorem 3 (Time and Space)**.** *For any deterministic* eNWA E *with state set* Q *defining a monadic query* P *and data tree* t*, the time complexity of our streaming algorithm with stack-and-state sharing to compute* $P(t)$ *is in* $\mathcal{O}(|E| + |Q|\ |t|)$ *and its space complexity in* $\mathcal{O}(|E| + depth(t)\ |Q| + C)$*, where* C *is the maximal number of buffered candidates of* P *on* t *at any event.*

Proof. By Proposition 9, the run-time per event is linear in the number of states Q. With the construction of E we thereby obtain an overall run-time complexity which is in $\mathcal{O}(|E| + |Q|\ |t|)$. Regarding space complexity, we have to keep E in memory all the time. The maximum number of buffered candidates at any event is C, while the DAG data-structure uses space in $depth(t)\ |Q|$. □

As we did in the time analysis, we can reduce $|Q|$ in our space analysis to the maximal number per event of states of the buffered candidates. Therefore, we can replace $|Q|$ by 2 in our practical experiments. So if the depth of the tree is bounded in addition (which is the case in our experiments), the space requirement is determined by the maximal number of buffered candidates per event, and by the size of the automaton. This will indeed be confirmed by our experiments in Figure 8.19.

8.7 QuiXPath tool

We present an implementation of our algorithms in the QuiXPath system and analyze its time and space performance experimentally.

8.7.1 Implementation, Tools, and Applications

We have implemented QuiXPath, which inherits a library, called Fxp tool, for compiling Fxp formulas to eNwas. QuiXPath is a streaming query evaluator for a fragment of XPath 3.0. The implementation consists of 3 parts. First, a streaming algorithm with stack-and-state sharing for answering eNwa queries, second, a compiler from Fxp to eNwas, and third a compiler from navigational XPath 3.0 to Fxp. All our implementation is done in Java 1.6, while relying on the free XPath parser from Saxon.

In addition to what we presented in this chapter, QuiXPath supports arithmetics, aggregation, and data comparisons, which is implemented on basis of networks of eNwas that we discuss in Chapter 10. Furthermore, there is a support for backwards axes, which are eliminated at the cost of forward axes and regular axes. Regular axes P^* are supported by QuiXPath, but only if P has 0-delay. This is enough for most backwards axes in the usual benchmarks. We would also like to notice that conditional regular axes are *not* sufficient for eliminating general backwards axes, as proven in [Ley 2009].

All our experiments are run on a machine which features an Intel Core i7-2720QM processor at 2.20GHz, 3.8GB of RAM, and an SSD hard drive. The operating system is a 64-bit version of ubuntu 12.04 LTS.

8.7.2 Benchmarks

We will analyze the coverage and the time and space performance of QuiXPath, and compare it to other streaming XPath evaluators, such as Spex, Saxon, and Gcx. For this, we need an XPath benchmark collection, see Section 5.11, which consists of a collection of XPath queries and a collection of documents, so that both of them can be scaled in size.

We mainly use the XPathMark benchmark collection[3] in its revised version (and not the one from the original paper [Franceschet 2005]). These are (parametric) queries about XMark documents that contain a table of bids and a table of bidders. The size of these documents can be scaled, basically by adding more bids and bidders. Amongst the XPathMark queries from Section 5.11 only queries A1-A8, B1-B7, B11(i)-B15(i), and C2 can be answered by some of the available tools (QuiXPath, Spex, Saxon, and Gcx). With $i = 1$ these are 39% of the XPathMark queries only. We added 2 further queries O1 and O2 to our collection, which are given in Figure 8.12. These queries do apply to the same XMark documents,

[3] http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html

```
O1: /site[closed_auctions/closed_auction/type]//item
O2: /site[c or not(c)]//bidder
```

Figure 8.12: Additional queries for illustrating the buffering behaviour.

but illustrate a different performance behavior since requiring to buffer much more candidates.

It should be noticed that neither Saxon nor Gcx support XPath directly, but through Xslt respectively XQuery programs. Therefore we could not run the queries from the XPathMark directly, but had to embed them in an Xslt or XQuery program. This kind of rewriting is not fully trivial, if one does not want the experiments to be biased by different output modes. For Spex we used their internal output mode "count_results" and for QuiXPath we counted the materialized answer nodes. For Saxon we use a style sheet which counts the number of query answers. For Gcx we were not able to count the number of answers without producing them. Therefore, we decided not to produce any output (rather than using their default output mode, which is to print the subtrees of the selected nodes).

There is a further problem when comparing to Gcx, which is that Gcx does not support Xml attributes. Since we did not want to take Gcx out of the race, we had to adapt the benchmark in the case of Gcx as follows: in documents, we replaced attributes by elements, and in queries we replaced attribute axes by child axes. These adaptations are not fully neutral. Therefore, we will also compare Gcx and QuiXPath on the same documents and queries in a separate study.

8.7.3 Performance Comparison

In Figure 8.13 we compare the different tools for their XPath "conceptual" coverage on the XPathMark benchmark: Gcx [Schmidt 2007], LNFA [Onizuka 2010], Saxon [Kay 2010], Spex [Olteanu 2007], XMLtk [Green 2004], XSeq [Mozafari 2012], and Xsq [Peng 2005]. Only 4 tools were available for testing in practice, but we can still compare the coverage in terms of the concepts that are supported, and also in the percentage of queries that require these features. When presenting such percentages, we restrict ourselves to the XPathMark queries with parameter $i = 1$. This is an advantage for our competitor tools, since these cannot deal with $n \geq 2$ anyway. Later on, we will also experiment with QuiXPath on queries with larger parameters $i \geq 2$ separately.

Only Gcx is able to deal with some queries with data joins such as C2. What is not covered by any of the tools are queries with data positions, full aggregations, and full negation. These features are used by the remaining 63% of queries of the XPathMark. Spex and QuiXPath cover mostly the same features except for top-level aggregation. Nevertheless, QuiXPath covers 37% of the XPathMark queries, while Spex can only deal with 22%. The other available tools cover even fewer queries, Gcx has 19% and Saxon is at 6%.

Name	QuiXPath	Gcx	LNFA	Saxon	Spex	XMLtk	XSeq	Xsq
Language	XPath	XQuery	XPath	Xslt	XPath	XPath	XSeq	XPath
Available	yes	yes	no	yes	yes	yes	no	no
Coverage of XPathMark	37%	19%	18%	6%	22%	5%	7%	11%
Downward axis	✓	✓	✓	✓	✓	✓	✓	✓
Complex filters	✓	✓	✓	✗	✓	✗	✗	✓
Backward axis	◡	✗	✗	✗	✓	✗	✗	✗
Negation	◡	◡	✗	✗	◡	✗	✗	✗
Top level aggreg.	✓	✓	✗	✓	✗	✗	✗	✓
Nested aggreg.	✗	◡	✗	✗	✗	✗	✗	✗
Arithmetics	✗	✗	✗	✓	✗	✗	✗	✗
0-delay reg. axis	✗	✗	✗	✗	✗	✗	✓	✗
Regular axis	✗	✗	✗	✗	✗	✗	✗	✗
Join	✗	◡	✗	✗	✗	✗	✗	✗
Multi inputs	✗	✗	✗	◡	✗	✗	✗	✗

supported: ✓ partially supported: ◡ not supported: ✗

Figure 8.13: System features.

The advantage in coverage of QuiXPath over Spex is mostly due to the fact that Spex does not support all queries that are produced by backward axes elimination. Saxon's low coverage for XPath in streaming mode may be motivated by their focus on Xslt. The coverage of Gcx is lowered since it does not support backward axes. While the 37% coverage of QuiXPath on the XPathMark benchmark is better than all previous tools, a much better result of 95% is achieved by λXP.

In Figure 8.14, we compare the runtime performance of QuiXPath against Spex, Saxon, and Gcx on the benchmark queries. A negative runtime here means that the query was not supported by the respective tool. The document was fixed to a 1.1GB XMark file. Generally, QuiXPath and Gcx show very good performance, outperforming Saxon and Spex.

The parametric queries B11(i)-B15(i) cannot be treated by neither Spex nor Gcx and also stresses QuiXPath to its limits. The problem comes with backwards axes, which must be rewritten into forward axes[4], such that the rewritten queries are supported by the query evaluator. Indeed, only QuiXPath is able to execute some of the rewritten queries, as we will discuss in Section 8.7.5.

In Figure 8.15 we give a more detailed comparison of the runtime performance of QuiXPath and Gcx on the same 1.2GB document without attributes with the queries adapted accordingly. It turns out that the parsing of Gcx is considerably higher than that of QuiXPath which inherits the parser from Saxon, even though

[4]In the general case, backwards axes elimination may need to introduce regular axes[Ley 2009], but not for the XPathMark queries considered here.

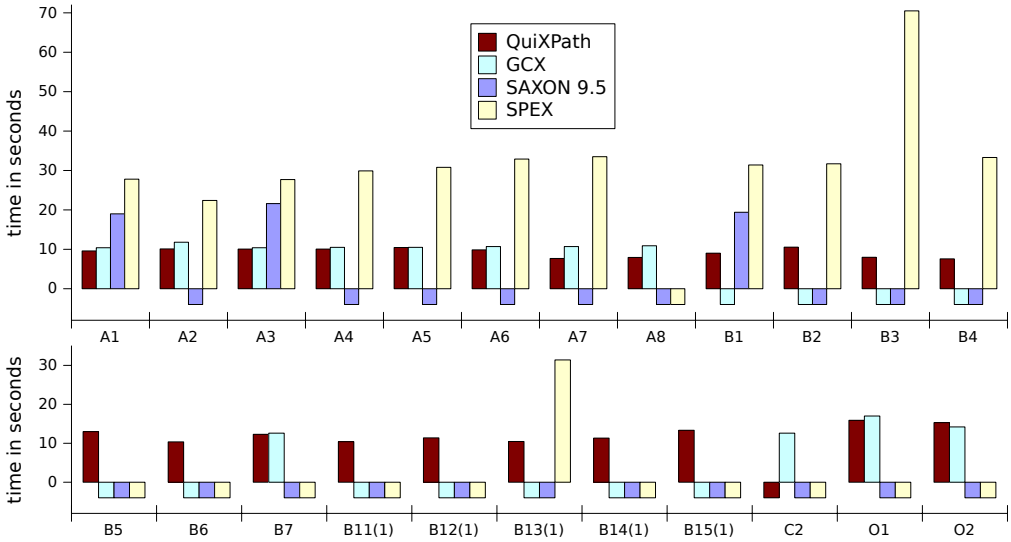


Figure 8.14: Runtime in seconds of QuiXPath, Spex, Saxon, and Gcx for the benchmark queries obtained on a 1.1GB XMark document. Whenever the query was supported, we depict a negative runtime. The runtimes were averaged over three runs, if there were no outliers (outliers were discarded).

Gcx is implemented in C++ and the others in Java. On the other hand side, the overhead of Gcx with respect to the parsing time is lower than that of QuiXPath.

8.7.4 Performance Analysis

In order to understand the time efficiency of QuiXPath on the different queries of the benchmark, it is essential to separate the parsing time from the time for pure query evaluation. This is relevant to all languages embedding XPath queries, since there, many XPath queries must be executed at the same time, while the document is parsed only once.

We propose to measure the *parsing-free evaluation time* $T_{parsefree}(P, d)$ of a query P on a document d as follows. We launch the query several times in parallel on the document, while parsing the document only once. When scaling the copy number, the parsing time should become irrelevant in the limit. Let $T(P, d, n)$ be the runtime that is needed to evaluate n copies of P in parallel on document d, and let $T_{parse}(d)$ be the parsing time for document d. We then define the parsing-free runtime $T_{parsefree}(P, d)$ of query P on document d as follows:

$$T_{parsefree}(P, d) = \lim_{n \to \infty} \frac{T(P, d, n) - T_{parse}(d)}{n}$$

The limit converged quickly for all our benchmark queries. Already for $n = 10$ the difference to the previous value was less than 50 ms for a 1.1 GB XMark document. In Figure 8.16 we display the parsing-free evaluation times for all benchmark queries

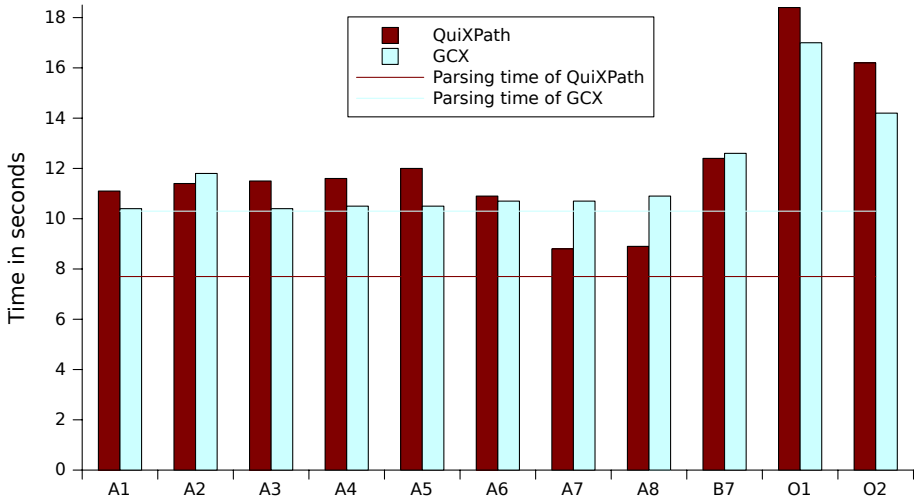


Figure 8.15: Runtime comparison of GCX and QUIXPATH for GCX compatible queries on a compatible 1.2GB XMark document.

on the 1.1 GB XMARK document, and related it to the overall runtime and to the parsing time.

The result shows that parser works in parallel with the query evaluator. The reason is that the CPU can work in parallel with file accesses to the stream. However, the degree of parallelism remains quite low for QUIXPATH. It should also be noticed that we were not able to compute the parsing-free evaluation time for the other tools, so that we do not know whether they exploit more parallelism or not.

Our next objective is to explain the difference in the parsing-free evaluation times for the various queries. The most relevant parameter is the workload of a query on a document, which is the number of pairs of states and non-projected events (see next paragraph), such that some buffered candidate was in this state at the event. Figure 8.17 shows that the workload correlates nicely to the parsing free evaluation time for all queries of our benchmark, except for queries that store big numbers of candidates, whose rejection or selection requires additional time.

The QUIXPATH implementation supports 2 kinds of projections (on which the workload depends) while we do not consider the projection algorithms in Chapter 9 that we have added only later in λXP. First, if a query does not depend on text data, all text events are projected away. This is indeed the case for all queries in the benchmark except for query C2 with data joins (that QUIXPATH cannot deal with). The second kind is depth projection for queries that do not use the descendant axis, so that they consider the nodes until a certain depth only. The queries with depth projection are exactly those with the lowest parsing-free evaluation time: A7, A8, B3, B4 (depth projection at depth 4), A6, B1 (depth projection at depth 5), and A1, A4 (depth projection at depth 7). The next best queries are those with 0-delay, since this reduces the workload too: A2, A3, B2, B6, B12(1), B13(1). The next class are queries without projection, but only few buffered candidates A5, B5, B7,

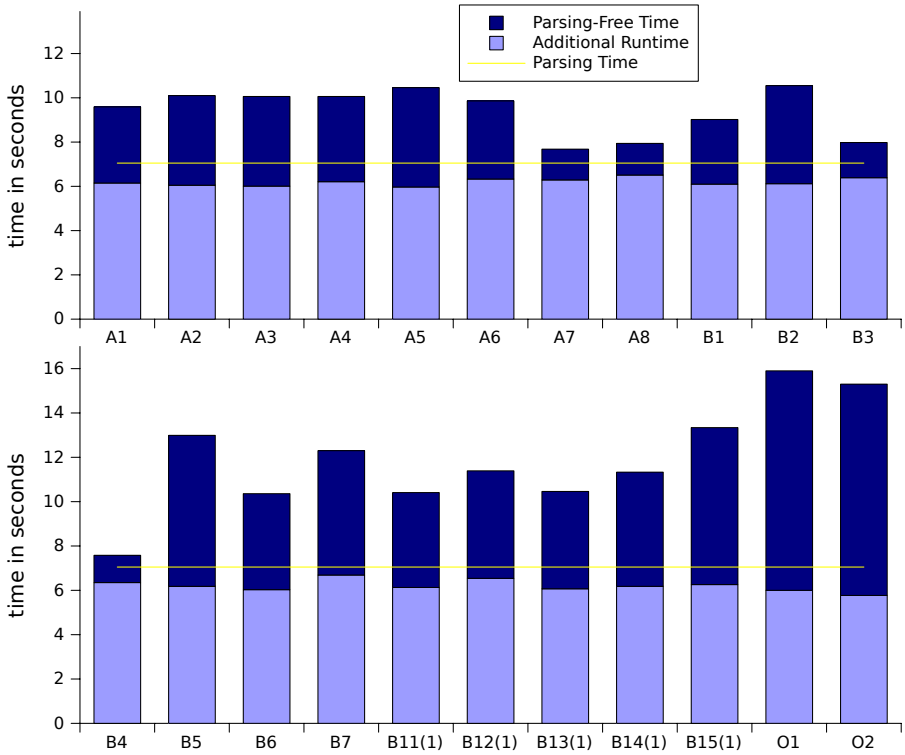


Figure 8.16: Parsing-free time of QuiXPath in relation to the additional runtime for the benchmark queries on the 1.1 GB XMark document.

B11(1), B14(1), and B15(1). The highest parsing-free evaluation time is obtained for the queries O1 and O2, which buffer thousands of candidates for many events, but all of them in the same state.

QuiXPath scales up to large document sizes. Thanks to stack and state sharing, the runtime remains linear in the size of the document as for all benchmark queries in Figure 8.18 on larger documents ranging from 1 GB until 28 GB in size. This even holds for queries such as O1 and O2, where the number of buffered candidates grows linearly. But since all of them are in the same state, the running times remain linear (and not quadratic). For document sizes greater than 17 GB the candidates needed to buffer for query O2 did not fit into memory, such that no evaluation times could be reported.

Query O2 is the only query from our benchmark collection, where QuiXPath is not earliest. In general QuiXPath is not earliest with the presence of valid respectively unsatisfiable filters (as in the examples showing the hardness of earliest query answering [Gauwin 2011a]). It was also proven in follow-up work [Lick 2013], that our early query answering algorithm is indeed earliest for a large subset of positive XPath queries.

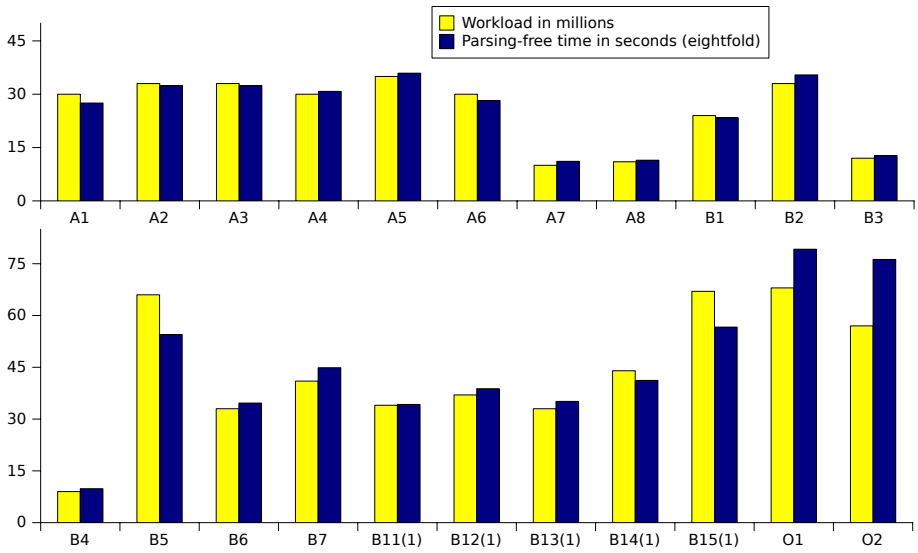


Figure 8.17: Workload compared to parsing-free evaluation time.

8.7.5 Detailed Analysis

We provide a detailed analysis of the performance of QuiXPath including measurements of its space consumption in particular.

We start with the analysis for the non-parametric XPathMark queries reported in Figure 8.19. The first step is to compile the XPath queries into simple Fxp formulas F, while eliminating backwards axes. The size of the formula and its conjunction width becomes apparent only after this precompilation phase. The maximal values are reached for the simple Fxp formula of query A8, with a size of 112 and a conjunction width of 12. These quite large values are due to backwards axes elimination and the following rewriting into *simple* Fxp formulas.

The next compilation step is to convert F into a descriptor of the nondeterministic eNwa. The number N of the states of this descriptor remains moderately small with at most 129 states for A8 and also the memory $\mathrm{KB}_{\mathrm{desc}}$ for its storage with at most 1186 KB. The number D of states visited by deterministic eNwa during on-the-fly instantiation and determinization also remains small with at most 127 for A8. The results confirm that the on-the-fly determinization inspects only a very small subset of the state space of the determinized automaton, and that the time required for on-the-fly instantiation and determinization (T_{det}) can be ignored in practice. The memory $\mathrm{KB}_{\mathrm{run}}$ needed for storing the deterministic automaton and the candidate buffer is small for all queries with few candidates with at most 672 KB for A8. It grows however linearly with the number of buffered candidates, as one can observe for queries O1 and O2.

The minimal memory required to evaluate a query in reasonable time is the sum of $\mathrm{KB}_{\mathrm{desc}}$ + $\mathrm{KB}_{\mathrm{run}}$ + 2000 KB, where the 2 MB serve for running the java virtual maschine itself. Our runtime experiments were performed with 3 GB of memory

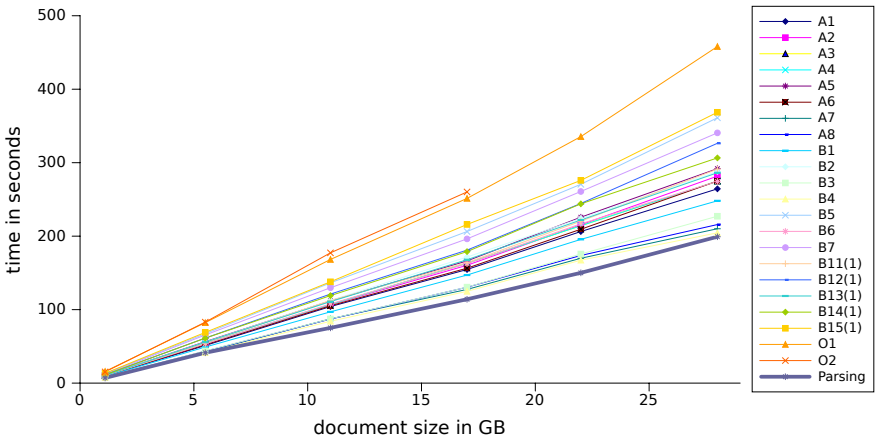


Figure 8.18: Runtimes of the benchmark queries for large documents.

in order to reduce overheads by repeated garbage collections. Less memory implies larger overheads. For query A1 we conducted a small experiment to estimate these overheads. We restricted the memory to 5, 15, 30, 60, and 120 MB and obtain as runtimes 14.0, 12.7, 10.7, 10.4, and 10.1 seconds respectively.

We next analyze the parametric queries B11(i)-B15(i). These contain backwards axes, which must be rewritten into forward axes. QuiXPath can do this, but the sizes of the Fxp formulas obtained by backwards axes elimination may be large, as shown in Figure 8.21. For instance, the formula for B14(4) has size 11804 and conjunction width 912. As a consequence, the corresponding eNwa descriptors will be huge, so that they quickly do no more fit into main memory (for $i \geq 5$ for all B13(i), B14(i), B15(i)), or may not leave enough space for on-the-fly determinization (for B13(4) and B15(4)).

Figure 8.20 shows the runtimes for the parametric queries that can still be evaluated by QuiXPath with 3 GB of memory. It should be noticed that the queries B11(i) are equivalent for all $i \geq 1$, and similarly for B12(i), B13(i), B14(i), and B15(i) respectively. Nevertheless, the sizes of the corresponding Fxp formulas grow with parameter i and also the conjunction width (since our compiler ignores query equivalence). Despite this, our compiler produces the same eNwa descriptor for all B11(i) independently of i, and similarly for all B12(i), while the eNwa descriptors for B13(i), B14(i), and B15(i) do grow quickly with i. The surprisingly good treatment of the families B11(i) and B12(i) can be explained as follows. The Fxp formulas obtained after backward axes elimination contain conjunctions of the same filter over and over (the filter `[./bidder]` for B11(i) and the filter `[@id]` for B12(i)). The automata for such conjunctions are independent of the number of times that the filter is conjoined with itself.

The queries B13(i), B14(i), B15(i) with $i \geq 3$ illustrate that the costs of on-the-fly determinization may become relevant for the runtime in extreme cases. Otherwise the runtime should be independent of the size of the eNwa descriptor, as explained earlier. In extreme cases, the deterministic automata have thousands of

	A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4
$\|F\|$	25	13	17	31	24	34	37	115	43	41	19	19
$w(F)$	0	0	0	1	1	2	2	12	0	2	0	0
N	17	9	12	27	17	35	21	129	24	39	13	13
KB_{desc}	201	102	131	237	177	307	212	1 186	276	411	136	141
D	14	10	14	27	23	32	23	127	14	49	16	15
KB_{run}	192	232	256	256	224	256	552	672	192	384	192	192
T_{det}	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.03	0.00	0.02	0.00	0.00

	B5	B6	B7	B11(1)	B12(1)	B13(1)	B14(1)	B15(1)	O1	O2
$\|F\|$	24	103	21	17	16	37	62	69	23	34
$w(F)$	1	4	1	1	0	2	0	3	1	2
N	13	58	15	12	10	32	33	34	25	18
KB_{desc}	161	571	122	120	99	389	326	372	251	193
D	32	28	20	17	11	28	19	29	25	12
KB_{run}	256	320	336	304	256	352	224	288	39 432	112 460
T_{det}	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.00

Figure 8.19: Detailed analysis for XPathMark queries. The queries are compiled to Fxp formulas F of size $|F|$ and conjunction width $w(F)$. N is the number of states of the eNwa descriptor constructed for F and KB_{desc} the size of the memory needed for its storage. D is the number of states of the deterministic eNwa constructed by on-the-fly instantiation and determinization. KB_{run} is the memory required at runtime for storing the deterministic eNwa and the candidate buffer, i.e., the overall memory needed is at least $\text{KB}_{\text{run}} + \text{KB}_{\text{desc}}$. T_{det} is the time in seconds needed to compute all D states by on-the-fly instantiation and determinization.

states (D), each of which contains thousands of states of the nondeterministic automaton (N). This is problematic since the determinization time is not linear but highly polynomial: overall it is in time $\mathcal{O}(D\ N^4\ |\Sigma|)$. Indeed, the overall time for on-the fly determinization (T_{det}) grows importantly in these cases. Furthermore, the backward axes elimination for B13(i), B14(i), and B15(i) produces Fxp formulas with many disjunctions, so that the candidates admit almost all of the N states of the nondeterministic eNwa. For this reason, the on-the-fly determinization requires high memory costs in these cases (included in KB_{run}), raising out-of-memory errors for B13(4) and B15(4).

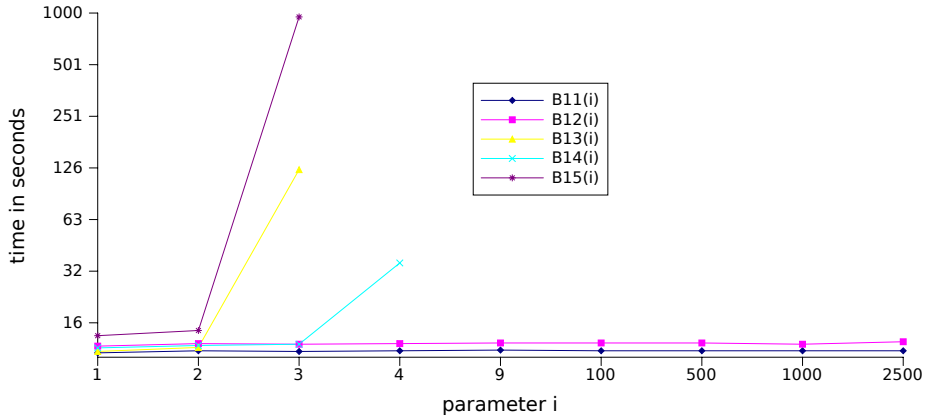


Figure 8.20: Runtime for parametric queries B11(i)-B15(i) of the XPathMark queryset on a 1.1 GB XMark document. When for some parameter i no runtime is displayed, then the corresponding eNwa descriptor did not fit into memory (for B14(5)) or it could not be evaluated on-the-fly (for B13(4) and B15(4)).

	B13(1)	B13(2)	B13(3)	B14(1)	B14(2)	B14(3)	B14(4)	B15(1)	B15(2)	B15(3)
$\|F\|$	37	173	863	62	374	2 116	11 804	69	527	4 149
$w(F)$	2	10	56	0	16	136	912	3	32	299
N	32	325	4 273	33	237	1 959	17 611	34	347	4 294
KB_{desc}	389	4 896	76 514	326	2 560	22 291	230 733	372	4 448	64 098
D	14	120	503	19	29	36	43	29	136	2 625
KB_{run}	352	1 683	32 747	224	384	1 821	16 868	288	1 448	394 780
T_{det}	0.01	0.76	113.02	0.00	0.02	1.13	21.05	0.00	0.54	971.88

Figure 8.21: Detailed analysis for the parametric queries B13(i)-B15(i). The backward axes elimination of QuiXPath infers a Fxp formula F of size $|F|$ and conjunction width $w(F)$ up to 912! The Fxp formulas are then compiled into eNwa descriptors with N states, which can be stored in a memory of size KB_{desc}. On-the-fly instantiation and determinization yields a deterministic eNwa of which D states are visited. The memory for storing this deterministic automaton and the canidate buffer is KB_{run}. T_{det} is the time in seconds needed for on-the-fly instantiation and determinization.

Chapter 9

Projection

Contents

In this chapter we develop projection algorithms for nested word automata. Our algorithms translate nested word automata to projection nested word automata that we introduce. They are a kind of mixed pushdown and counting automata. The projection nested word automata is then evaluated over a stream producing a projected nested word on which it runs. Our experiments show that we gain a speed up of a factor of 4 in parsing-free time with respect to our query answering algorithm for navigational XPath queries by early nested word automata.

9.1 Introduction

Projection is most relevant for efficient Xml processing algorithms, as shown for in-memory evaluators for XQuery in [Marian 2003] and for a fragment of XPath in [Maneth 2010]. The projection algorithm for XQuery runs in Saxon [Kay 2004], today's most used Xml processing tool.

The objective of this chapter is to initiate the development of projection algorithms for processing Xml streams. Given that a single program written in one of the Xml standards XQuery, Xslt, or XProc contains a collection of XPath queries, we are interested in the evaluation of a collection of XPath queries on a single input stream. The parsing time can be shared between many XPath queries, and thus should be counted seperately. Therefore, we are mainly interested in the

parsing-free time for query evaluation. Note however, that the parsing-free time for a single query is often dominated by the parsing time.

We will restrict ourselves to projection for navigational XPath queries, since these are fundamental to all others. For instance, in order to check whether the root of a tree has at least 5 a-children, all other children of the root can be projected. The computation of the projection still requires to read the entire input tree, but the time for this can be shared similarly to the parsing time. As discussed before in Chapter 8, navigational XPath queries are reduced to running an Nwa on all possible answer candidates.

Projection for finite automata is well known [Frisch 2004, Maneth 2010]. It amounts to project away all letters of the input word that do not change the state. Projection for Nwas is more tedious, since such automata have a stack by which they can pass information from opening tags to corresponding closing tags. Therefore, one cannot simply project an opening tag away without taking care of the corresponding closing tag. Our idea is that a projected nested word should contain jump symbols $.^{i}.$ for projected factors, where the integer i stands for the excess of the factor, i.e., the difference between the number of opening and closing tags. We present *projection nested word automata (*PNwas*)*, a kind of mixed pushdown and counting automata, that input projected nested words which beside others contain integers as letters. These integers allow the automaton to compute the depth of the current node of the tree at any time, and also the excess of the last jump. Conversely, a projection of a nested word with respect to a given Nwa can be computed by any corresponding PNwa. It may be surprising, but it turns out there may exist different PNwas with maximal projection for the same Nwa. Therefore, our projection algorithm has to make its choices.

We then lift Nwa projection to the evaluation of navigational XPath queries on Xml streams. It turns out that the parsing-free time for our query answering algorithm of Chapter 8 is reduced by a factor of 4 on average on the usual XPathMark benchmark.

9.2 A Nested Word Automaton

Let Σ be a finite alphabet. Let P_Σ be the set of parenthesis with labels in Σ, that is the set of opening tags $\langle a\rangle$ and closing tags $\langle /a\rangle$ where $a \in \Sigma$. We consider complete nested words over P_Σ, such that corresponding opening and closing tags have the same label. Furthermore we omit internal letters Δ for what follows. As a result Xml data trees and their linearizations contain no data values, which is a strong simplification. However the treatment of data values works orthogonal to the treatment of element nodes, since they can be encoded into elements.

Let us consider the unranked tree in Figure 9.1. The Xml stream obtained by linearizing this unranked tree into a complete nested word is given in Figure 9.2. A deterministic Nwa that accepts the nested word is given in Figure 9.3. It defines the XPath filter `[//a/b]` which accepts all Xml trees that contain some

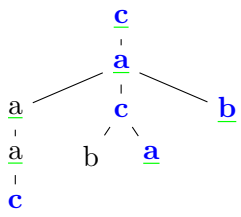


Figure 9.1: An unranked tree.

⟨c⟩	⟨a⟩	⟨a⟩	⟨a⟩	⟨c⟩	⟨/c⟩	⟨/a⟩	⟨/a⟩
		⟨c⟩	⟨b⟩	⟨/b⟩			
			⟨a⟩	⟨/a⟩	⟨/c⟩		
		⟨b⟩	⟨/b⟩	⟨/a⟩	⟨/c⟩		

Figure 9.2: The corresponding nested word is an XML stream.

a-descendant with a b-child. Rules containing label sets represent sets of rules, one for each label.

9.3 Projection NWAs

We next introduce projected nested words. Let $\mathbb{N}_0 = \mathbb{N} \uplus \{0\}$ be the by 0 extended set of natural numbers, and $\mathbb{Z}$ the set of integers. For any unranked tree, we are interested in the binary node relations child ch, descendant ch^+, n-th grand parents ch^{-n} where $n \in \mathbb{N}$, descendants of n-th grand parents ch^{-n}/ch^+, children of n-th grand parents ch^{-n}/ch, and stay at *self*. So let:

$$Rels = \{ch, ch^+, ch^{-n}, ch^{-n}/ch^+, ch^{-n}/ch, self \mid n \in \mathbb{N}\}.$$

9.3.1 Projected Nested Words

A *projected nested word* is a word whose letters are jump symbols $.^{\overset{i}{\cdot}}.$ where $i \in \mathbb{Z}$ and jump targets $p@r$ where $p \in P_\Sigma$ and $r \in Rels$. We write $P_{\Sigma}^{\overset{\cdot}{\cdot}}$ for the set of all these letters. We assume that any jump target is proceeded by a jump symbol that indicates the excess of the jump, that is the depth difference in the tree or equivalently, the difference of the numbers of opening and closing tags in the complete nested word. We also assume that projected nested words are well-nested up to jumping.

Two examples for projected nested words are given in Figure 9.4. Both are valid descriptions of the nested word in Figure 9.2: pw_1 projects to the letters drawn in blue, while pw_2 projects to the letters drawn in green. As we will see,

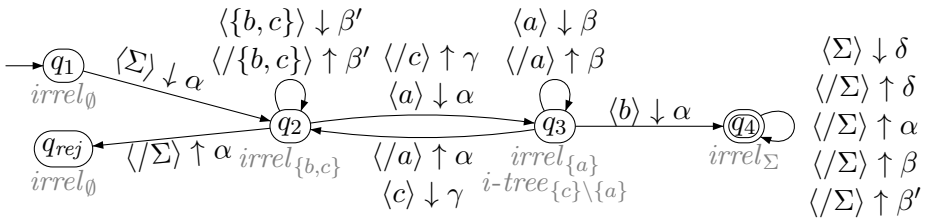


Figure 9.3: A deterministic NWA over $\Sigma = \{a, b, c\}$ for XPATH filter `[//a/b]`.

pw_1: for all a-nodes without an a-parent and all non-a-children of a-nodes keep the opening and closing tags, until the opening tag of the first match of `//a/b`:

$$\langle\mathbf{c}\rangle \,.^{0}_{:}.\, \langle\mathbf{a}\rangle@ch^{+} \,.^{2}_{:}.\, \langle\mathbf{c}\rangle@ch^{+} \,.^{0}_{:}.\, \langle/\mathbf{c}\rangle@self \,.^{-2}_{:}.\, \langle\mathbf{c}\rangle@ch^{-3}/ch^{+}$$
$$.^{0}_{:}.\, \langle\mathbf{a}\rangle@ch^{+} \,.^{0}_{:}.\, \langle/\mathbf{a}\rangle@self \,.^{0}_{:}.\, \langle/\mathbf{c}\rangle@ch^{-1} \,.^{0}_{:}.\, \langle\mathbf{b}\rangle@ch^{-1}/ch^{+}$$

pw_2: for all a-nodes and all b-children of a-nodes keep the opening and closing tags, until the opening tag of the first match of `//a/b`:

$$\langle\mathbf{c}\rangle \,.^{0}_{:}.\, \langle\mathbf{a}\rangle@ch^{+} \,.^{0}_{:}.\, \langle a\rangle@ch^{+} \,.^{0}_{:}.\, \langle a\rangle@ch^{+} \,.^{0}_{:}.\, \langle/a\rangle@self$$
$$.^{0}_{:}.\, \langle/a\rangle@ch^{-1} \,.^{1}_{:}.\, \langle\mathbf{a}\rangle@ch^{-1}/ch^{+} \,.^{0}_{:}.\, \langle/\mathbf{a}\rangle@self \,.^{-1}_{:}.\, \langle\mathbf{b}\rangle@ch^{-2}/ch$$

Figure 9.4: Two projected nested words describing the nested word in Figure 9.2.

both projections can be obtained from the NWA in Figure 9.3. Note that the initial opening tag is always kept for technical reasons. Except of this, both projections are maximal, in that no further tags can be projected away: they just preserve enough information for deciding whether the original nested word satisfies the filter `[//a/b]`. Nevertheless, none of these two projections is more general than the other. The green projection pw_2 has the advantage to keep only tags with letters occurring in the XPATH filter `[//a/b]`. The blue projection pw_1, has the advantage to keep fewer of these tags, but therefore, it also keeps some others.

The blue projection pw_1 starts with $\langle\mathbf{c}\rangle$, meaning that any matching nested word must start with $\langle c\rangle$. The next factor $.^{0}_{:}.\, \langle\mathbf{a}\rangle@ch^{+}$ describes a nested word with excess 0 that is followed by $\langle a\rangle$ in descendant position, i.e., by the opening tag of an a-child of the root. The next factor $.^{2}_{:}.\, \langle\mathbf{c}\rangle@ch^{+}$ describes a nested word with excess 2 followed by $\langle c\rangle$ opening a descendant. Then $.^{0}_{:}.\, \langle/\mathbf{c}\rangle@self$ requires to jump with excess 0 to the closing tag $\langle/c\rangle$ of the same node. Next, $.^{-2}_{:}.\, \langle\mathbf{c}\rangle@ch^{-3}/ch^{+}$ asks to jump with excess -2 to an opening tag $\langle c\rangle$ of a descendant of a grand-grand-grand-parent, etc.

9.3.2 Projection Nested Word Automata

We introduce PNWAS as a mixture of a pushdown and a counting automaton, that receive projected nested words as input. The counting serves for updating the depths of nodes when jumping, so that the depth of the current node can always

be deduced from the current stack. Whenever jumping over a projected factor, the excess of this factor is pushed. This is an integer that is popped when trying to close the jump.

Definition 14. *A* PNWA *is a tuple* $A = (\Sigma, Q, Q_I, Q_F, \Gamma, R)$ *like an* NWA *but with different kinds of transition rules: given* $a \in \Sigma$, $\gamma \in \Gamma$, *and* $q, q' \in Q$, *there are rules of the following types in* R, *for changing the state from* q *to* q'.

Open: $q \xrightarrow{\langle a \rangle \downarrow \gamma} q'$ *Like for* NWA*s.*

Close: $q \xrightarrow{\langle / a \rangle \uparrow \gamma} q'$**.** *Like for* NWA*s.*

Jump to a child or a descendant: $q \xrightarrow[\forall z \geq 0]{\stackrel{z}{\cdots}\langle a \rangle @ r \downarrow z \downarrow \gamma} q'$**, where** $r \in \{ch, ch^+\}$**.**

When $r = ch$ *then* z *must be* 0*, and we jump to the opening tag of an* a*-child and push first* 0 *and then* γ *onto the stack. When* $r = ch^+$ *then we jump over* z *descendants to the opening tag of an* a*-descendant, and push first* z *and then* γ *onto the stack. For short we denote this transition as* $q \xrightarrow{ju(\langle a \rangle, r, \gamma)} q'$.

Rejump to another child or descendant: $q \xrightarrow[\forall z, z'.\ z' \geq 0, z + z' \geq 0]{\stackrel{z}{\cdots}\langle a \rangle @ ch^{-(z'+1)} / r \uparrow z' \downarrow z + z' \downarrow \gamma} q'$**,**
where $r \in \{ch, ch^+\}$**.**

While trying to close a jump from some grand parent to some node one can rejump to another opening a*-tag of a child or a descendant of the same grand parent. The excess of the jump to the first node* z' *on the stack is updated to the excess of the second node* $z + z'$*. Furthermore,* γ *is pushed. For short, we write this transition as* $q \xrightarrow{reju(\langle a \rangle, r, \gamma)} q'$.

Jump to the closing tag of the self node: $q \xrightarrow{\stackrel{0}{\cdots}\langle / a \rangle @ \textbf{\textit{self}} \uparrow \gamma} q'$**.** *Jump to the closing tag of the self* a*-node. In this case,* γ *is popped from the stack.*

Jump back to the jump's origin: $q \xrightarrow[\forall z \geq 0]{\stackrel{-z}{\cdots}\langle / a \rangle @ ch^{-(z+1)} \uparrow z \uparrow \gamma} q'$**.** *When trying to close a jump, one may jump back to the closing tag of the* a*-node where the current jump started. The excess of* $-z$ *is popped from the stack together with the symbol* γ *which was pushed for the non-jumped* a*-node. For short we write* $q \xrightarrow{ju\text{-}back(\langle / a \rangle, \gamma)} q'$.

Close last jump step: $q \xrightarrow[\forall z > 0]{\langle / a \rangle \uparrow z \downarrow z-1} q'$**.** *When trying to close a jump, one may read a closing* a*-tag for which the corresponding opening* a*-tag was jumped, so that no stack symbol was pushed. In this case the excess of the jump on the stack must be updated from* z *to* $z - 1$.

A configuration of a PNWA is a pair in $Q \times (\Gamma \uplus \mathbb{N}_0)^*$ consisting of a state in Q and a stack in $(\Gamma \uplus \mathbb{N}_0)^*$. A run r of a PNWA A on a projected nested word over Σ is a function that maps any prefix of the projected nested word to a configuration. The run must start in some configuration with some initial state and the empty

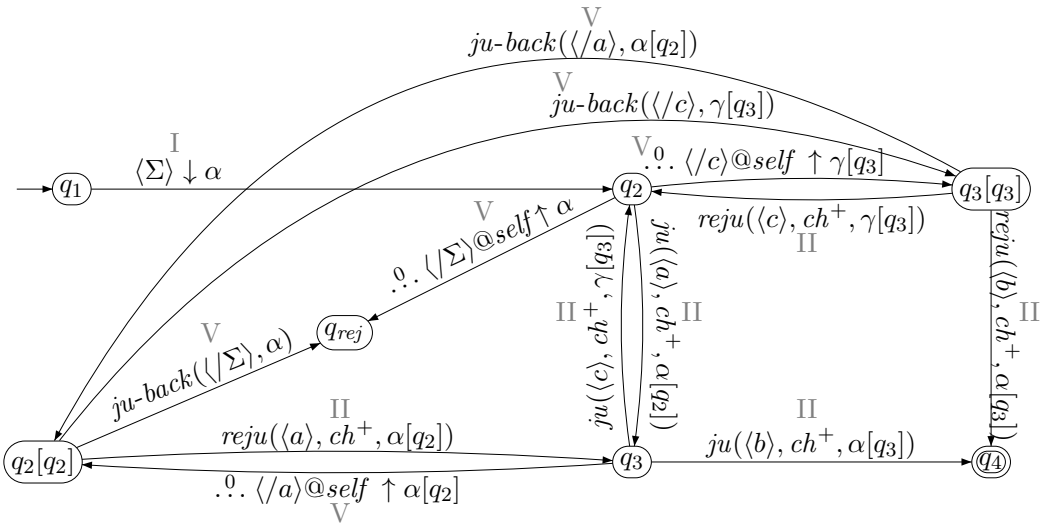


Figure 9.5: PNwa $\mathbf{A}_1$ for the XPath filter `[//a/b]`.

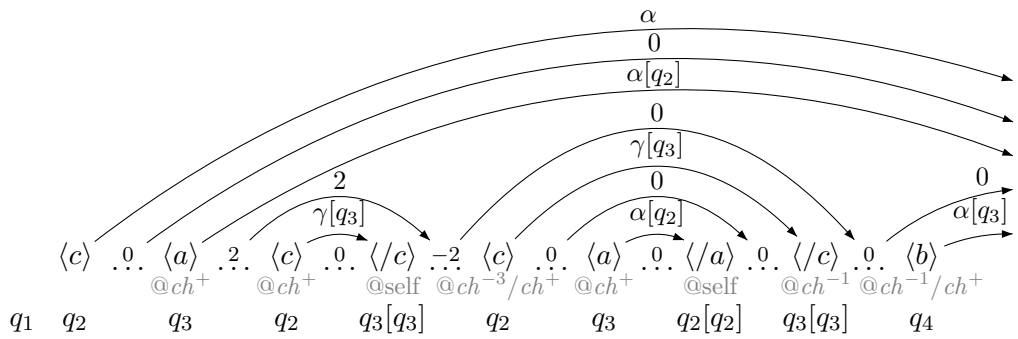


Figure 9.6: A successful run of the PNwa $\mathbf{A}_1$ of Figure 9.5 on $pw_{\mathbf{1}}$.

stack, i.e., $r(\varepsilon) \in Q_I \times (\Gamma \uplus \mathbb{N}_0)^*$. Furthermore, for any prefix wl where $l \in P_\Sigma^{\cdot\cdot}$, the configuration $r(w)$ must be transformed into $r(wl)$ by applying some rule consuming letter l. A run on a projected nested word w is called successful if it eventually reaches a configuration with a final state, i.e., if $r(w') \in Q_F \times (\Gamma \uplus \mathbb{N}_0)^*$ for some prefix w' of w. The language $\mathcal{L}(A)$ of a PNwa A is the set of all projected nested words that permit a successful run on A.

9.3.3 Examples of PNwas and Runs

First Projection. In Figure 9.5 we present PNwa $\mathbf{A}_1$ that is a projection of the Nwa in Figure 9.3 for the XPath filter `[//a/b]`. This automaton accepts the blue projection $pw_{\mathbf{1}}$ in Figure 9.4 of the nested word in Figure 9.2. Automaton $\mathbf{A}_1$ visits the opening and closing tags of all a-nodes with no a-parent, and of all non-a-children of these a-nodes, and jumps over all other nodes. Automaton $\mathbf{A}_1$ accepts when the first match of `[//a/b]` arrives. In Figure 9.6, we illustrate a successful

run of $\mathbf{A}_1$ on pw_1. The states of configurations are placed below the tags, while the stack consists of the labels on the subedges above the state. Edges between tags indicate their correspondence. Furthermore there are edges for jumps to children and descendants, where the excess is pushed, while jumps to the jump origin close the jump, and rejumps update the excess on the stack. The only exceptions are jumps to the closing tag of self nodes, where no excess is pushed. In general PNWA $\mathbf{A}_1$ works as follows. It starts in the initial state q_1, it opens the root and goes into q_2, where either the root can be closed to q_{rej} or where it can jump over b and c nodes to the opening tag of an a-descendant and go to q_3. There are 3 possibilities depending on what happens first: (1) close the a-node, and go to $q_2[q_2]$, (2) jump down over a sequence of a-nodes to the opening tag of a c-descendant and go to q_2, or (3) jump down over a sequence of a-nodes to the opening tag of a b-descendant and accept in q_4. In state q_2 a c-node with a sequence of a-grand parents can be closed to $q_3[q_3]$. The sequence of a-grand parents consists of a sequence of jumped a-nodes and one not jumped a-node π at the top. Continuing depending on what comes first, the following can happen in $q_3[q_3]$: (1) jump back to the closing tag of the a-grand parent π and go to $q_2[q_2]$, (2) rejump over a sequence of a-nodes, while staying below π, to the opening tag of a c-descendant and go to q_2, or (3) rejump over a sequence of a-nodes, while staying below π, to the opening tag of a b-descendant, and accept in state q_4. In state $q_2[q_2]$ there are 3 possibilities depending on what happens first: (1) rejump over a sequence of b and c nodes to the opening tag of an a-descendant and goto q_3, while staying below a not-jumped c-grand parent with a sequence of a-grand parents if exists, (2) jump back to the closing tag of a not-jumped c-grand parent with the a-grand parents sequence if exists, or else (3) close the root to q_{rej}.

Second Projection. A second projection of the NWA from Figure 9.3 is given by the PNWA A_2 in Figure 9.7. This one accepts the green projection pw_2 in Figure 9.4. A successful run of A_2 on pw_2 from Figure 9.4 is illustrated in Figure 9.8.

Recomputing Stack Symbols. We give an PNWA that contains rules to recompute stack symbols. In Figure 9.9 we give a deterministic NWA for XPATH filter `[//a/following-sibling::b]`. A PNWA $\mathbf{A}_3$ for the same query is depicted in Figure 9.13. A successful run of $\mathbf{A}_3$ on projected nested word pw_3 from Figure 9.12 is illustrated in Figure 9.14.

9.3.4 Evaluation of Projection Nwas

Next we are interested to evaluate a collection of PNWAS obtained from deterministic NWAS on a single complete nested word. For this, we need to project the nested word with respect to the PNWAS, and run the PNWAS on the respective projected nested word. Therefore, we have to define how to project a nested word w with respect to a deterministic PNWA. More generally, we define a projection $\pi_q(w)$ for

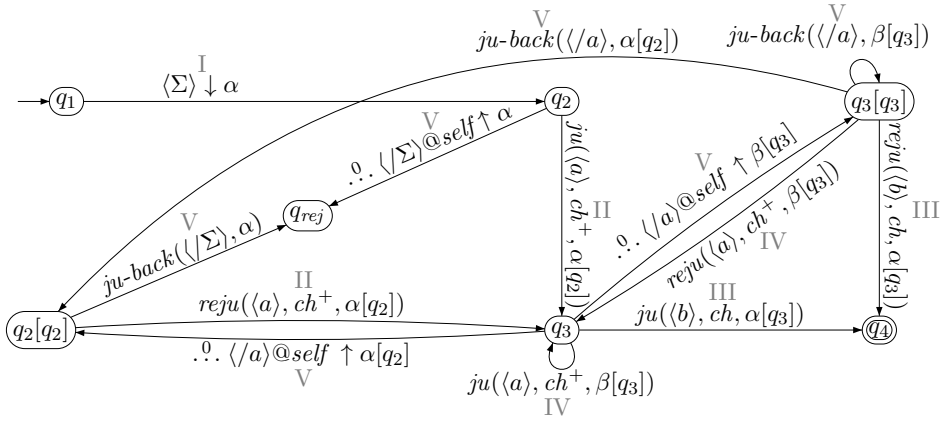


Figure 9.7: PNWA A_2 for the XPATH filter `[//a/b]`.

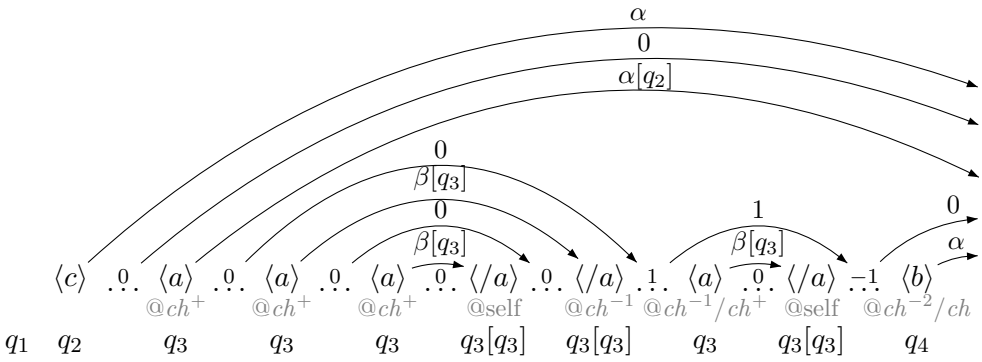


Figure 9.8: A successful run of the PNWA A_2 of Figure 9.7 on pw_2.

any suffix w of some nested word in P_Σ^* and state q of a PNWA A:

$$\pi_q(w) \;= \overset{i}{\cdot\cdot}\, p@r\, \pi_{q'}(w'')$$

such that $w = w'pw''$ for some $p \in P_\Sigma$ and $w', w'' \in P_\Sigma^*$, where w' is the shortest prefix, so that there exists a rule of A from q to q' consuming letter $p@r$ for some $r \in \mathit{Rels}$, and i is the excess of w'.

9.4 Irrelevant Labels and Prefixes of Nested Words

In this section, we define properties of NWA states which allow to skip parenthesis with irrelevant labels and irrelevant prefixes of nested words, that is prefixes of linearizations of subtrees.

Definition 15. *An* NWA E *can jump over parenthesis with labels in* L *and incoming state* q *– in formulas* $q \in irrel_L$ *– if there exists a stack symbol* γ*, such that* E *has all transitions shown in Figure 9.15, no other opening transition pushing* γ*, no other* L*-opening transition in* q*, and no other* L*-closing transition with* γ *in* q*.*

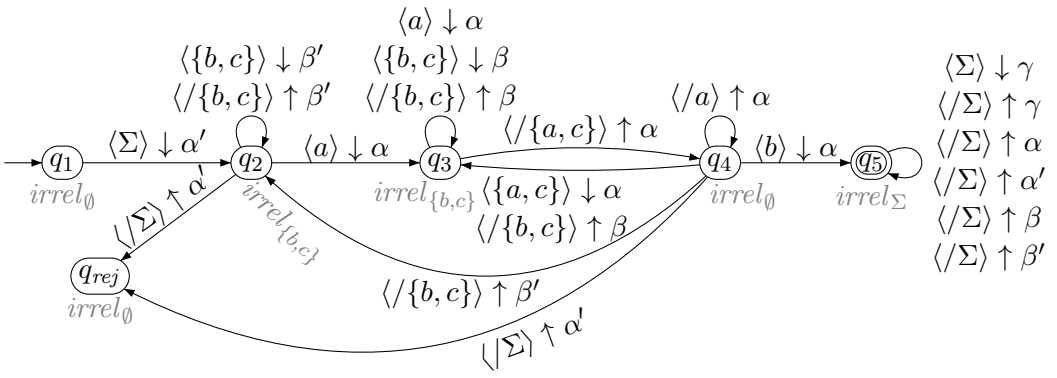


Figure 9.9: A deterministic NWA over $\Sigma = \{a, b, c\}$ for XPATH filter `[//a/following-sibling::b]`.

If $q \in \mathit{irrel}_L$ then any sequence of letters in P_L is irrelevant in state q, so that it can be removed from the nested word and replaced by a jump symbol. Consider a run of E on a nested word w and assume $q \in \mathit{irrel}_L$. We next argue, that we can replace all letters in P_L of w with ingoing state q by jump symbols, while "repairing" the run. The first point is that the state is not changed when reading such letters, so that their removal keeps the states correct. But we must also take care of the stack. If an opening tag $\langle a\rangle$ is removed but not the corresponding closing tag, then we have to repair the run, in order to be able to reproduce the missing stack symbol when needed. The idea is to memoize the state before jumping. Since this state does not change while jumping, one can then recompute the stack symbol that was pushed for any letter that was jumped over. Conversely, it is not possible that a closing tag $\langle/a\rangle$ was removed but not the corresponding opening tag, since the symbol pushed at $\langle a\rangle$ must be γ, and by definition of $q \in \mathit{irrel}_L$ there is no other opening transition pushing γ than that started in q.

Definition 16. *An* NWA *E in state q can jump over prefixes of nested words (subtrees) that start in $\langle L\rangle$, do not contain letters in $P_{L'}$, and either end with the closing tag of the subtree's root or with a letter in L', if there exist three different stack symbols $\gamma, \gamma', \gamma''$ and a state q' such that the transitions shown in Figure 9.16 exist, but no further opening transitions with γ, no further transitions with γ', and no further opening transitions in q' for L', and no further closing transition in q for L popping γ. In this case, we write $q \in \mathit{i\text{-}tree}_{L\setminus L'}$ and call q a state of irrelevant subtrees.*

In the easiest case where $q \in \mathit{i\text{-}tree}_{L\setminus\emptyset}$ one can jump over nested words linearizing subtrees, with incoming state q and labels in L only. When opening the root of the subtree, the state changes to q' and stays there until closing the root and going back to q. So the removal of the subtree does not change the state globally. In this case, the full nested word of the subtree is read, so the stack difference is zero. In the case where $L' \neq \emptyset$ it is more tricky to repair the run, in order to deal with missing stack symbols. But it remains possible, since the state used within the subtree does

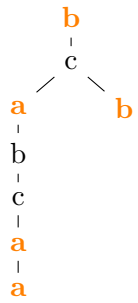


Figure 9.10: Another unranked tree.

$$\langle \mathbf{b}\rangle\langle c\rangle\langle \mathbf{a}\rangle\langle b\rangle\langle c\rangle\langle \mathbf{a}\rangle\langle \mathbf{a}\rangle\langle /\mathbf{a}\rangle\langle /\mathbf{a}\rangle\langle /\mathbf{c}\rangle\langle /b\rangle\langle /\mathbf{a}\rangle\langle \mathbf{b}\rangle\langle /b\rangle\langle /c\rangle\langle /b\rangle$$

Figure 9.11: The nested word that corresponds to the unranked tree in Figure 9.10.

$$\langle \mathbf{b}\rangle \ .^{1}_{\cdot}. \ \langle \mathbf{a}\rangle @ ch^{+} \ .^{2}_{\cdot}. \ \langle \mathbf{a}\rangle @ ch^{+} \ .^{0}_{\cdot}. \ \langle \mathbf{a}\rangle @ ch^{+} \ .^{0}_{\cdot}. \ \langle /\mathbf{a}\rangle @ self$$
$$.^{0}_{\cdot}. \ \langle /\mathbf{a}\rangle @ ch^{-1} \langle /\mathbf{c}\rangle \ .^{0}_{\cdot}. \ \langle /\mathbf{a}\rangle @ ch^{-2} \langle \mathbf{b}\rangle$$

Figure 9.12: Projected nested word pw_3 describing the nested word in Figure 9.11.

not change, so that it can be memoized and so that missing stack symbols can be recomputed at closing time.

For illustration, we have annotated the state of the NWA in Figure 9.3 with the properties that they satisfy. It turns out that state q_3 satisfies both properties $irrel_{\{a\}}$ and $i\text{-}tree_{\{c\}\setminus\{a\}}$, but that we cannot perform the two corresponding projections at the same time. When choosing projection with $irrel_{\{a\}}$ we obtain the PNWA $\mathbf{A}_1$ from Figure 9.5, and when choosing projection with $i\text{-}tree_{\{c\}\setminus\{a\}}$ we obtain the second projection PNWA A_2 from Figure 9.7.

9.5 Projection from Nwas to PNwas

We show how to project deterministic NWAs E to a PNWA A. For any state q of E, we chose a projection property $choice(p)$, which is either $irrel_L$ or $i\text{-}tree_{L\setminus L'}$ for some sets $L, L' \subseteq \Sigma$. Note that $irrel_\emptyset$ can always be assigned, so that this assumption can always be satisfied, but not always in a unique manner.

Any state of A is either a state of q of E or a pair of states of E that we write as $q[q']$. Such a pair means that one is in state q and that on the top of the stack is a jump symbol i that was pushed from a jump over i descendants that started in state q'. Any stack symbol of A is either a stack symbol γ of E or a pair written as $\gamma[q]$ of a stack symbol and a state of E. γ serves as the stack symbol that was pushed before at opening tags, while q is the state where a previous jump started. Whenever such a pair $\gamma[q]$ is on the stack then the symbol below is always a jump

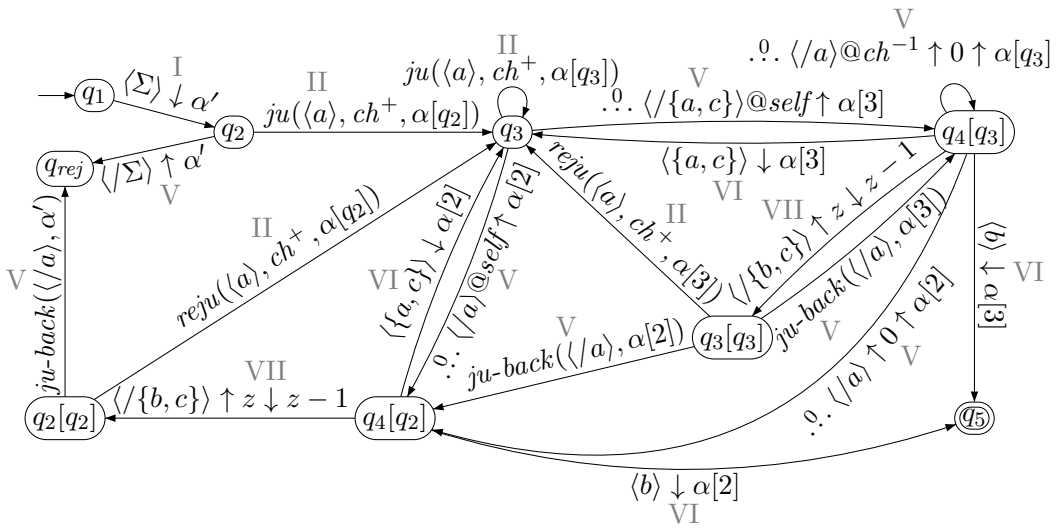


Figure 9.13: PNwa $\mathbf{A}_3$ for the XPath filter `[//a/following-sibling::b]`.

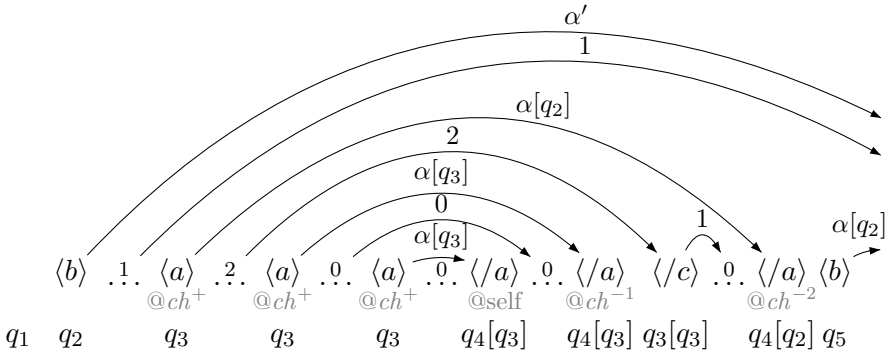


Figure 9.14: A successful run of the PNwa $\mathbf{A}_3$ of Figure 9.7 on pw_3 from Figure 9.12.

symbol i that was pushed by a jump over i descendants that started in state q. The sets of initial and final states remain unchanged.

Every transition rule of E gives rise to a possible empty set of transition rules of A, according to rules I–VII in Figure 9.17. In PNwa $\mathbf{A}_1$ from Figure 9.5 we annotated transitions accordingly. Transitions from an initial state are translated to non-jumping transitions that open the root. If $choice(q) = irrel_L$, then all looping transitions required by $irrel_L$ are removed. The other opening transitions starting from q are translated to jumping and rejumping transitions to descendants and descendants of grand parents. If $choice(q) = i\text{-}tree_{L\setminus\emptyset}$ then the opening and closing L transitions, and looping transitions required by $i\text{-}tree_{L\setminus\emptyset}$ are removed. The other opening transitions starting from q are translated to jumping and rejumping rules to children and children of grand parents. If $choice(q) = i\text{-}tree_{L\to L'}$ then the opening and closing L transitions, and looping transitions required by $i\text{-}tree_{L\to L'}$ are

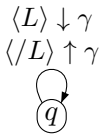


Figure 9.15: $q \in \mathit{irrel}_L$.

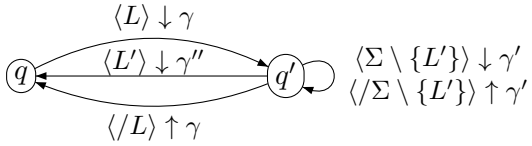


Figure 9.16: $q \in \mathit{i\text{-}tree}_{L \setminus L'}$.

removed. All other transitions with opening tag $a \in L'$ departing q are translated to jumping and rejumping rules for descendants. Closing transitions are translated to six rules: Two rules to close self nodes, two rules to jump back to jump's origins, and two last rules that close parents in a state $q[q'']$ for $q \neq q''$. Those states do not allow to rejump, since the previous jump started in a different state q'' than the current state q, and therefore they also do not allow to jump back to the jump's origin. For these states opening and closing transitions are translated as indicated, while recomputing stack symbols, that have not been pushed for jumped grand parents.

Proposition 10 (Soundness)**.** *Let E be a deterministic* NWA *E with initial state q_0 and A be a* PNWA *obtained from E by our projection algorithm. It then holds for any nested word w that $w \in \mathcal{L}(E)$ if and only if $\pi_{q_0}(w) \in \mathcal{L}(A)$.*

9.6 Node Selection

Node selection as needed for typical XPATH queries requires some extensions to what we have presented so far, which we will discuss in the following.

Let us consider the unranked tree with the corresponding nested word in Figure 9.18. The tree represents an XML library, in which a single node has a label which is annotated by a fixed variable x, meaning that this node is to be tested for selection by an NWA that answers a given XPATH query. Such x-annotations can be computed in streaming mode as discussed in Chapter 8.

The NWA in Figure 9.19 defines the XPATH query `library/book/title`, which selects all *title*-nodes in a tree, whose parent is a *book*-node, and whose grand parent is the *library*-root. It accepts the nested word of Figure 9.18, where the title tag is annotated, and thereby selects the corresponding title node in the library without the x annotation. For sake of completeness, we assume a sink state q_{rej} in the NWAs, which allows for early rejection: whenever reached by a deterministic NWA

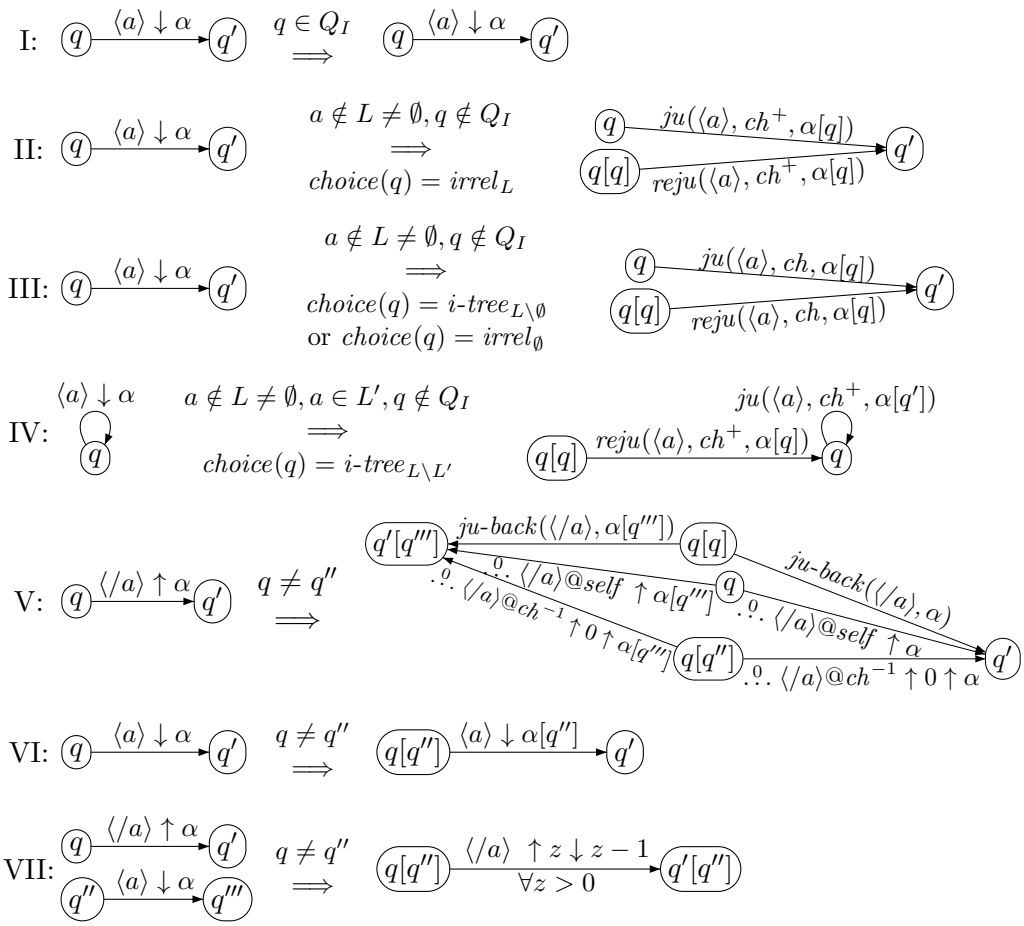


Figure 9.17: Rewriting system for rules of a deterministic NWA to rules of the PNWA.

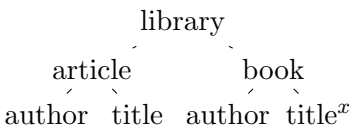
the run will fail on the completed tree. Transitions with annotated labels going to q_{rej} allow then to reject candidates that will fail early.

Due to the alphabet with annotated labels $\Sigma = \{a, a^x \mid a \in \Sigma^{no\text{-}var}\}$, where $\Sigma^{no\text{-}var}$ is a signature without variable annotations, we need to adapt Definitions 15 and 16 of Section 9.4.

Definition 15 becomes

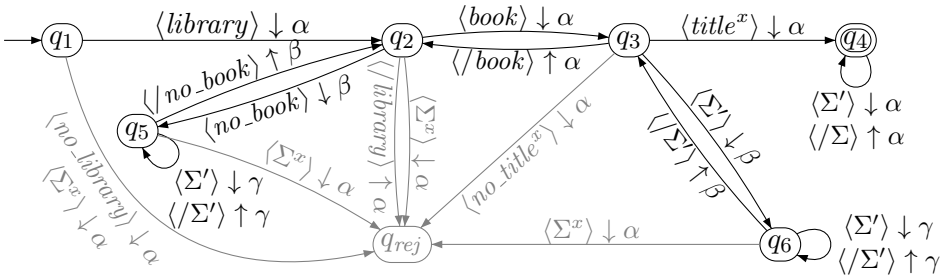
Definition 17. *An* NWA *E can jump over parenthesis with labels in $L \subseteq \Sigma^{no\text{-}var}$ and incoming state q, if there exists a stack symbol γ and a sink state q_{rej}, such that E has all transitions shown in Figure 9.20, no other opening transition pushing γ, no other L-opening transition in q, no other $\{a^x \mid a \in L\}$-opening transition in q, and no other L-closing transition with γ in q.*

There we require transitions with letters a^x with $a \in L$ go to rejection state q_{rej}. This is because since, otherwise if there exists a transition with an annotation that does not go to the rejection, letter a becomes relevant. For these letters candidates are created that may be selected in some future. Similarly Definition 16 requires an



⟨library⟩⟨article⟩⟨author⟩
⟨/author⟩⟨title⟩⟨/title⟩⟨/article⟩
⟨book⟩⟨author⟩⟨/author⟩⟨title^x⟩
⟨/title^x⟩⟨/book⟩⟨/library⟩

Figure 9.18: An XML tree library and the corresponding nested word.



$\Sigma = \{a, a^x \mid a \in \Sigma'\}$, $\Sigma' = \{\mathit{library}, \mathit{book}, \mathit{author}, \mathit{title}, \mathit{article}\}$, $\Sigma^x = \Sigma \setminus \Sigma'$,
$\mathit{no\_book} = \Sigma' \setminus \{\mathit{book}\}$, $\mathit{no\_library} = \Sigma' \setminus \{\mathit{library}\}$, $\mathit{no\_title}^x = \Sigma^x \setminus \{\mathit{title}^x\}$.

Figure 9.19: An NWA for XPATH query `library/book/title`.

adaptation, where transitions with letters $\{a^x \mid a \in L\}$ and $\{a^x \mid a \in L'\}$ for L, L' of Figure 9.16 must go into the rejection state.

9.7 Experiments

We implemented NWA projection within the QUIXPATH system and tested it on the (revised) XPATHMARK query set of Section 5.11 on queries A1-A8, B1-B7, and B11(1)-B15(1).

As argued in the introduction, it is most natural to measure the efficiency in *parsing-free time* as presented in Section 8.7.4.

In a first experiment, we start from FXP and enhance it with projection. The results are presented in Figure 9.21 for a 559 MB XPATHMARK document. It turns out that projection reduces the parsing-free running time for this query set by a factor of 4.3, which is a major improvement, in particular when evaluating many XPATH queries in parallel as needed for streaming XSLT or XQUERY programs. In our second experiment, we compare the overall running time of our PNWA evaluator of XPATH queries on XML streams with SAXON's in-memory evaluator [Kay 2004]. For each of our queries, we compare the full running times including parsing, when evaluating the query n-times. The results are given in Figure 9.22. It turns out that QUIXPATH with projection for NWAs can answer on average a query up to 12 times in parallel, in no more time than needed by SAXON for the same task.

One observes that running less than 12 queries in parallel with PNWAS is a lot quicker than running them with SAXON, mostly due to the expensive in-memory

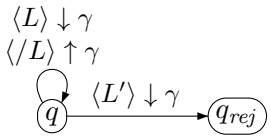


Figure 9.20: $L \subseteq \Sigma^{no\text{-}var}$ and $L' = \{a^x \mid a \in L\}$.

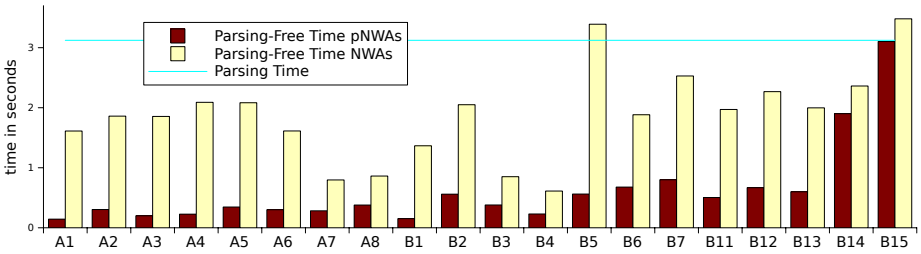


Figure 9.21: Improvement by NWA projection of XPATH query evaluation.

tree creation. But when running more than 12 queries on small documents, the advantage of in-memory evaluation takes over. Indeed, without the time for parsing and in-memory tree construction, SAXON in-memory evaluation is still faster by a factor 20 in average than streaming with PNWAS. With the improvements of the present paper, it now seems possible that stream processing can become more efficient than in-memory evaluation in practice in the future.

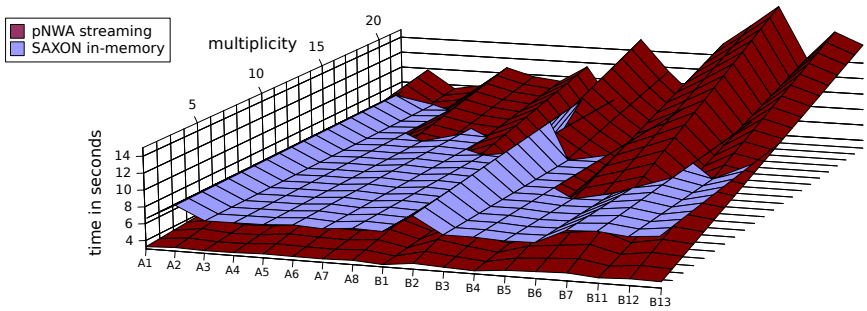


Figure 9.22: Streaming wins vs. in-memory evaluation for up to 12 queries in parallel.

Part III

Query Evaluation Algorithms for Full XPath

We present an algorithm for evaluating λXP queries on data trees that are stored in-memory. When composed with our compiler from XPath 3.0 to λXP, this yields an in-memory evaluator for full XPath.

In contrast to more standard λ calculi, the difficulty for obtaining an in-memory evaluator for λXP is that it is unclear how to formulate a small-step semantics for this language, given that it also supports first-order logical queries. For instance, for evaluating $\exists x.x \in \{1,2,3\} \wedge x+1 \geq 2$ one has to generate and test all possible values for x and apply the addition function to these values in order to test whether they satisfy $x+1 \geq 2$. Our evaluator will rely on the mode system for λXP, in order to properly split the logical and programming parts of λXP queries. The logical parts will be evaluated by generating answer sets, while the programming part will be evaluated by executing function applications.

We will then show how to turn our in-memory evaluator into a streaming evaluation algorithm. For this we need to split the navigational from non-navigation aspects of λXP queries. Purely navigational λXP queries will be evaluated by running early nested word automata as described earlier. General λXP queries are decomposed into networks of navigational λXP queries. These can be evaluated on Xml streams by running networks of early nested word automata on them.

Chapter 10

In-Memory Evaluation of λXP Queries

Contents

In this chapter we present how to evaluate in-memory a network of navigational λXP expressions. The network of navigational λXP is defined by simple expressions that we introduce. For the evaluation of these formulas we present algorithms that compute their semantics and variable assignments according to the mode system of λXP that we introduced earlier. We thus evaluate well-moded and well-typed λXP expressions.

10.1 Simplification of λXP Queries

λXP formulas can be decomposed into a network of navigational formulas, i.e. a λXP formula where all subexpressions below an axis are navigational. This is a simplication, because the evaluation of these simple kind of λXP formulas can be split into an evaluation of navigational and non-navigational formulas, such that an evaluator for navigational formulas can be lifted to an evaluator for arbitrary λXP expressions.

Another simplication is the elimination of backward axes. This is neccessary for a streaming evaluator that cannot go back on the stream, but also useful for an in-memory evaluator, for example, since no support for backward axes is required.

10.1.1 Simple Queries

We are interested in simple λXP formulas, where all formulas E in axis-expressions $A(E)$ are navigational, where below negations no non-deterministic constructs occur, and where goto-nodes formulas **at** no non-navigation expression exist.

Definition 18. *We call an* λXP *formulas* E simple*, if*

1. *for all formulas* $A(E')$ *in* E*, where* A *is an axis,* E' *contains no non-navigational formulas,*
2. *for all navigational formulas* $A(F)$ *in* E*, where* A *is an axis,* F *contains no negations* $\neg F'$*, where* F' *contains disjunctions or non-child axes, and*
3. *for all formulas* $\boldsymbol{at}(E_1, E_2)$ *in* E*,* E_2 *is navigational.*

For example, for query

$$
\begin{array}{ll}
(Q_1) & \mathbf{at}(x_1, \{x \mid ch^+(\texttt{tweet} \wedge is_x \\
& \quad \wedge \exists^=(\texttt{fn:data}(set2list(\{x' \mid ch(\texttt{user\_name} \wedge is_{x'})\})), \\
& \qquad \mathbf{at}(x_2, \texttt{fn:data}(set2list(\\
& \qquad\quad \{x' \mid ch^+(\texttt{students} \wedge ch(\texttt{name} \wedge is_{x'}))\}))))\}))
\end{array}
$$

discussed already in Section 6.2.10, we give the following equivalent simple query, where all subqueries of axes terms are navigational.

$$
\begin{array}{ll}
(Q'_1) & \{x \mid \exists y[\mathbf{at}(x_1, ch^+(\texttt{tweet} \wedge is_x \wedge is_y)) \\
& \quad \wedge \exists^=(\texttt{fn:data}(set2list(\{x' \mid \mathbf{at}(y, ch(\texttt{user\_name} \wedge is_{x'}))\})), \\
& \qquad \texttt{fn:data}(set2list(\\
& \qquad\quad \{x' \mid \mathbf{at}(x_2, ch^+(\texttt{students} \wedge ch(\texttt{name} \wedge is_{x'})))\}))))]\}
\end{array}
$$

10.1.2 Decomposition into Networks of Navigational Queries

We next show that any λXP formulas can be translated into a simple λXP formula, i.e. a network of navigational queries. In particular, this allows to lift the streaming evaluator of Chapterchap:enwa for navigational λXP expressions to a general streaming evaluator for λXP in Chapter 11.

Lemma 11. *Any* λXP *formula can be transformed into a simple* λXP *formula.*

Proof. Let E be an λXP formula.

Conditions (1) of definition 18 can be obtained as follows. We identify maximal navigational contexts C in E with formulas $E_1, \ldots, E_n$ in the holes: $C(E_1, \ldots, E_n)$, where all E_i are non-navigational and have an axis A above them in C for all $1 \leq i \leq n$. Then we replace such $C(E_1, \ldots, E_n)$ in E by $\exists y_1 \ldots \exists y_n.[C(is_{y_1}, \ldots, is_{y_n}) \wedge \mathbf{at}(y_1, E_1) \wedge \ldots \wedge \mathbf{at}(y_n, E_n)]$.

Negations $\neg$ below an axis in navigational formulas of E which contain disjunctions or non-child axes, are eliminated as follows. We identify maximal navigational contexts C in E with formulas $\neg E_1, \ldots, \neg E_n$ in the holes: $C(\neg E_1, \ldots, \neg E_n)$, where

all E_i are navigational, containing a disjunction or some non-child axes, and where all E_i have an axis A above them in C for all $1 \leq i \leq n$. Then we will replace such $C(\neg E_1, \ldots, \neg E_n)$ in E by $\exists y_1 \ldots \exists y_n.[C(is_{y_1}, \ldots, is_{y_n}) \wedge \neg\mathbf{at}(y_1, E_1) \wedge \ldots \wedge \neg\mathbf{at}(y_n, E_n)]$.

Last but not least, we consider formulas $\mathbf{at}(E_1, E_2)$ in E, where E_2 is not navigational. We rewrite $\mathbf{at}(E_1, E_2)$ as follows: Let $E_2 = E'(E'_1, \ldots, E'_n)$, where E'_i are navigational formulas and E' does not contain any axes or label tests, assuming condition (1) is true. Then $\mathbf{at}(E_1, E_2)$ is rewritten to $E'(\mathbf{at}(E_1, E'_1), \ldots, \mathbf{at}(E_1, E'_n))$. This is correct and can be proven by induction on the structure of non-navigational formulas E_2. The idea is to push the **at** formula down to navigational formulas, who depend directly on the evaluation start node, changed by **at**-formulas.

□

Simple query Q'_1 can be obtained from Q_1 following the proof of the lemma.

10.1.3 Backward Axes Elimination

We show how to evaluate backward axes in simple navigational queries. Let F be a simple navigational λXP formula, which contains backward axes, such as parent par, ancestor par^+, or preceding-sibling ps^+. Since we are considering networks of navigational queries, queries are evaluated from different nodes in general. We are thus considering the more general case of an evaluation of F at some arbitrary node π, instead of evaluating F from some root node always.

Our backward axes elimination procedure rewrites the query by "pushing up" backward axes in the formula, until they appear at the root position of the formula. This is useful for queries that are evaluated at the root of a tree, since there no parent, no ancestor, and no preceding-sibling exists. Instead of rewriting formulas F, which is evaluated at some node π, we rewrite backward axes in the slightly adapted formulas:

$$start(F, z) =_{def} ch^*(is_z \wedge F)$$

Evaluating F at node $\pi \in \text{nodes}(s)$ is then equivalent to evaluating $start(F, z)$ on the root $root(\pi)$ to which π belongs:

$$[\![F]\!]_{s,\pi,\mu} = [\![start(F, z)]\!]_{s,root(\pi),\mu \uplus [z/\pi]}$$

Therefore the elimination process "pushes up" backward axes and disjunctions in $start(F, z)$, until at the root of the term backward axes can be eliminated, by applying the rewrite rules that we give in Figure 10.1 exhaustively. Notice that since $start(F, z)$ is simple we can assume that no backward axes appear below a negation, for which reason the set of given rewrite rules suffices.

Backward axes below existential axes:

$$\begin{array}{lcl}
ch(par(F) \wedge F') & = & F \wedge ch(F') \\
ch(par^+(F) \wedge F') & = & (F \vee par^+(F)) \wedge ch(F') \\
ch(ps^+(F) \wedge F') & = & ch(F \wedge ns^+(F')) \\
ch^+(par(F) \wedge F') & = & (F \wedge ch(F')) \vee ch^+(F \wedge ch(F')) \\
ch^+(par^+(F) \wedge F') & = & ((par^+(F) \vee F) \wedge ch^+(F')) \vee (ch^+(F \wedge ch^+(F'))) \\
ch^+(ps^+(F) \wedge F') & = & ch^+(F \wedge ns^+(F')) \\
ns^+(par(F) \wedge F') & = & par(F) \wedge ns^+(F') \\
ns^+(par^+(F) \wedge F') & = & par^+(F) \wedge ns^+(F') \\
ns^+(ps^+(F) \wedge F') & = & ((ps^+(F) \vee F) \wedge ns^+(F')) \vee ns^+(F \wedge ns^+(F'))
\end{array}$$

Disjunctions where A is an axis, and De Morgan's law:

$$\begin{array}{lcl}
A(F \vee F') & = & A(F) \vee A(F') \\
(F \vee F') \wedge F'' & = & (F \wedge F'') \vee (F' \wedge F'')
\end{array}$$

Eliminate backward axes at the root of the term:

$$\begin{array}{lcl}
par(F) & = & \mathit{false} \\
par^+(F) & = & \mathit{false} \\
ps^+(F) & = & \mathit{false}
\end{array}$$

Figure 10.1: Backward axes rewrite rules.

Example.

$$\begin{array}{lcl}
ch(is_x \wedge (par^+(a) \vee ps^+(b))) & = & ch((is_x \wedge par^+(a)) \vee (is_x \wedge ps^+(b))) \\
& = & ch((is_x \wedge par^+(a))) \vee ch(is_x \wedge ps^+(b))) \\
& = & ((a \vee par^+(a)) \wedge ch(is_x)) \vee ch(b \wedge ns^+(is_x)) \\
& = & ((a \vee \mathit{false}) \wedge ch(is_x)) \vee ch(b \wedge ns^+(is_x)) \\
& = & (a \wedge ch(is_x)) \vee ch(b \wedge ns^+(is_x))
\end{array}$$

10.2 Evaluation Algorithm

What we would like to compute is the semantics $[\![E]\!]_{s,\pi,\alpha}$ for any expression E, sequence of trees s, node π, and variable assignment α. We restrict ourselves to expressions E that are well-moded and well-typed. This permits us to separate logical subexpressions, where variables assignments must be enumerated, from program subexpressions, where functions are applied.

10.2.1 Specification

In order to deal with set constructions and existential quantifiers, we have to compute variable assignments for well-moded expressions $E \triangleright V_1 \to V_2$ of type **bool**.

More precisely, given an assignment of the variables in V_1 we have to compute an extension to the variables in $V_1 \cup V_2$ so that E becomes true. For well-moded expressions of other types, the set V_2 will always be empty.

Lemma 12. *If $E \triangleright V_1 \to V_2$ with $V_2 \neq \emptyset$ and $E : T$ then $T :$ **bool**.*

If $V_2 = \emptyset$ then we are interested in a program evaluator, which inputs some V_1 assignment and computes the value of E for it. Otherwise, we have $T =$ **bool** and are interested in a query evaluator that inputs some V_1 assignment, and generates the set of all extensions to variables in V_2 so that E becomes true.

Proposition 11. *Let E be an expression with mode $E \triangleright V_1 \to V_2$ and type $E : T$.*

a) There exist a program evaluator A that computes the value of E for any sequence s of trees, $V_1 \cup V_2$ assignments α into s-values, and any node $\pi \in nodes(s)$:

$$A(s, \alpha, \pi) = [\![E]\!]_{s,\pi,\alpha}$$

under the assumption that a program evaluator exists that computes the semantics of expressions $c \in \mathcal{C}onst$.

b) There exists a query evaluator B that for any sequence s of trees and V_1 assignment to s-values computes the set of $V_1 \cup V_2$ assignments to s-values that extend β and satisfies E:

$$B(s, \beta, \pi) = \begin{cases} \bot & \text{, if } A(s, \alpha, \pi) = \bot \\ & \text{for some } \alpha \supseteq \beta \\ \{\alpha \mid \beta \subseteq \alpha,\ A(s, \alpha, \pi) = \mathit{true}\ \} & \text{, otherwise} \end{cases}$$

Proof. The construction of A and B is by simultaneous induction on derivations of judgments $E \triangleright V_1 \to V_2$. We first note that if $V_2 = \emptyset$ then we can always obtain B from A as follows:

$$B(s, \pi, \beta) \quad = \quad \begin{cases} \{\beta\} & , A(s, \pi, \beta) = \mathit{true} \\ \emptyset & , A(s, \pi, \beta) = \mathit{false} \\ \bot & , A(s, \pi, \beta) = \bot \end{cases}$$

Submodes. Suppose that the judgement $E \triangleright V_1 \to V_2$ is derived by the first rule for submodes:

$$\frac{E \triangleright V \to V'}{E \triangleright V \cup V'' \to V'}$$

Let A', B' be the program and query evaluators for $E \triangleright V \to V'$, respectively. We then define program and query evaluators for $E \triangleright V \cup V'' \to V'$ as follows:

$$\begin{array}{lcl} A(s, \pi, \alpha) & = & A'(s, \pi, \alpha_{|_{V \cup V'}}) \\ B(s, \pi, \beta) & = & \begin{cases} \bot & \text{, if } B'(s, \pi, \beta_{|_V}) = \bot \\ \{\beta \uplus \beta' \mid \beta' \in B'(s, \pi, \beta_{|_V})\} & \text{, otherwise} \end{cases} \end{array}$$

Next consider the case, where the judgement was inferred by the second rule for submodes:

$$\frac{E \triangleright V \to V' \cup V''}{E \triangleright V \cup V'' \to V'}$$

Let A', B' be the evaluators for $E \triangleright V \to V' \cup V''$. We define the evaluators A, B for $E \triangleright V \cup V'' \to V'$ as follows:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & A'(s,\pi,\alpha) \\ B(s,\pi,\beta) & = & \begin{cases} \bot & \text{, if } B'(s,\pi,\beta_{|_V}) = \bot \\ \{\beta' \mid \beta' \in B'(s,\pi,\beta_{|_V}),\ \beta \subseteq \beta'\} & \text{, otherwise} \end{cases} \end{array}$$

Conjunction. The mode is inferred by inference rule:

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \cup V' \to V'' \quad E = E_1 \wedge E_2}{E \triangleright V \to V' \cup V''}$$

Let A_1, B_1 be evaluators for $E_1 \triangleright V \to V'$ and A_2 and B_2 evaluators for $E_2 \triangleright V \cup V' \to V''$. We define evaluators for $E \triangleright V \to V' \cup V''$ by induction:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & A_1(s,\pi,\alpha) \wedge^{Bool^\bot} A_2(s,\pi,\alpha) \\ B(s,\pi,\beta) & = & \begin{cases} \bot & \text{, if } B_1(s,\pi,\beta) = \bot \text{ or} \\ & \quad B_2(s,\pi,\beta_1) = \bot \text{ for} \\ & \quad \text{some } \beta_1 \in B_1(s,\pi,\beta) \\ \{\beta_2 \mid \beta_1 \in B_1(s,\pi,\beta), \beta_2 \in B_2(s,\pi,\beta_1)\} & \text{, otherwise} \end{cases} \end{array}$$

Disjunction.

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \to V' \quad E = E_1 \vee E_2}{E \triangleright V \to V'}$$

Let A_i and B_i be evaluators for $E_i \triangleright V \to V'$ for $i \in \{1, 2\}$. We define evaluators for $E \triangleright V \to V'$ as:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & A_1(s,\pi,\alpha) \vee^{Bool^\bot} A_2(s,\pi,\alpha) \\ B(s,\pi,\beta) & = & B_1(s,\pi,\beta) \cup^\bot B_2(s,\pi,\beta) \end{array}$$

Negation.

$$\frac{E \triangleright V \to \emptyset}{\neg E \triangleright V \to \emptyset}$$

Let A' be the program evaluator for $E \triangleright V \to \emptyset$. We define A and B for $\neg E \triangleright V \to \emptyset$ as follows:

$$A(s,\pi,\alpha) \quad = \quad \neg^{Bool^\bot} A'(s,\pi,\alpha)$$

The modes are such that no variables are generated, so that $B(s,\pi,\beta)$ can be defined from $A(s,\pi,\beta)$.

Constants.

$$\frac{true}{c \triangleright \emptyset \to \emptyset}$$

We define program evaluator A for $c \triangleright \emptyset \to \emptyset$ as:

$$A(s,\pi,\alpha) \quad = \quad [\![c]\!]_{s,\pi}$$

According to the proposition there exists a program evaluator computing $[\![c]\!]_{s,\pi}$. Again, no variables are generated, so that $B(s,\pi,\beta)$ can be defined from $A(s,\pi,\beta)$.

Variables.

$$\frac{true}{x \triangleright \{x\} \to \emptyset}$$

We define evaluators A and B for $x \triangleright \{x\} \to \emptyset$ as:

$$A(s,\pi,\alpha) \quad = \quad \alpha(x)$$

Due to the modes, no variables are generated, $B(s,\pi,\beta)$ can be defined from $A(s,\pi,\beta)$.

Existential quantification.

$$\frac{E \triangleright V \to V' \cup \{x\} \quad x \notin V'}{\exists x.E \triangleright V \to V'}$$

Let B' be a query evaluator for $E \triangleright V \to V' \cup \{x\}$. We define evaluators for $\exists x.E \triangleright V \to V'$ as follows:

$$A(s,\pi,\alpha) \quad = \quad \begin{cases} \bot & , B'(s,\pi,\alpha_{|_V}) = \bot \\ true & , \exists \beta' \in B'(s,\pi,\alpha_{|_V}) : \alpha \subset \beta' \\ false & , \text{otherwise} \end{cases}$$

$$B(s,\pi,\beta) \quad = \quad \begin{cases} \bot & , \text{if } B'(s,\pi,\beta) = \bot \\ \{\beta'_{|_{V \cup V'}} \mid \beta' \in B'(s,\pi,\beta)\} & , \text{otherwise} \end{cases}$$

Navigation formulas. Consider first the rule for navigation by axes:

$$\frac{E \triangleright V \to V'}{A(E) \triangleright V \to V'}$$

Let A' and B' be evaluators for $E \triangleright V \to V'$. We define evaluators for $A(E) \triangleright V \to V'$ as:

$$A(s,\pi,\alpha) \quad = \quad (\exists \pi' \in \text{nodes}(s) : A^s(\pi,\pi') \wedge^{Bool_\bot} A'(s,\pi',\alpha) = true)$$

$$B(s,\pi,\beta) \quad = \quad \begin{cases} \bot & , \text{if } B'(s,\pi',\beta) = \bot \text{ for} \\ & \text{some } \pi' \text{ with } A^s(\pi,\pi') \\ \{\beta' \mid A^s(\pi,\pi'), \beta' \in B'(s,\pi',\beta)\} & , \text{otherwise} \end{cases}$$

Node tests.

$$\frac{true}{L \triangleright \emptyset \to \emptyset}$$

Since no variables are generated we only need to define the program evaluator A:

$$A(s,\pi,\alpha) \quad = \quad (\pi \in L^s)$$

Negated navigational formulas.

$$\frac{E \triangleright V \to V' \quad \text{E is navigational}}{\neg E \triangleright V \to V'}$$

Let A' and B' be evaluators for $E \triangleright V \to V'$. We define evaluators A and B for $\neg E \triangleright V \to V'$ as:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & (\neg^{Bool_\perp} A(s,\pi,\alpha)) \\ B(s,\pi,\beta) & = & \left\{ \begin{array}{ll} \perp & \text{, if } B'(s,\pi,\beta) = \perp \\ \{\beta' \mid \beta \subseteq \beta',\ \beta' \notin B'(s,\pi,\beta) & \text{, otherwise} \\ \qquad \beta' \text{ is } V \cup V' \text{ assignment to } s\} & \end{array} \right. \end{array}$$

Notice that β' in the definition of $B(s,\pi,\beta)$ only permits variables of type **node**, since E is navigational. Therefore there is only a finite number nodes to try for β'.

Location.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to V' \quad E = \mathbf{at}(E_1, E_2)}{E \triangleright V \to V'}$$

Let A_1 be a program evaluator for $E_1 \triangleright V \to \emptyset$, and A_2 and B_2 evaluators for $E_2 \triangleright V \to V'$. We define evaluators for $E \triangleright V \to V'$ as follows:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & \left\{ \begin{array}{ll} A_2(s, A_1(s,\pi,\alpha), \alpha) & \text{, } A_1(s,\pi,\alpha) \neq \perp \\ \perp & \text{, otherwise} \end{array} \right. \\ B(s,\pi,\beta) & = & \left\{ \begin{array}{ll} B_2(s, A_1(s,\pi,\beta), \beta) & \text{, } A_1(s,\pi,\beta) \neq \perp \\ \perp & \text{, otherwise} \end{array} \right. \end{array}$$

Conditional.

$$\frac{\begin{array}{c} E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to V' \quad E_3 \triangleright V \to V' \\ E = \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \end{array}}{E \triangleright V \to V'}$$

Let A_1 be a program evaluator for $E_1 \triangleright V \to \emptyset$ and A_i and B_i be evaluators for $E_i \triangleright V \to V'$ for $i \in \{2,3\}$. We define evaluators A and B for $E \triangleright V \to V'$ as:

$$\begin{array}{rcl} A(s,\pi,\alpha) & = & \left\{ \begin{array}{ll} A_2(s,\pi,\alpha) & \text{, } A_1(s,\pi,\alpha) = \mathit{true} \\ A_3(s,\pi,\alpha) & \text{, } A_1(s,\pi,\alpha) = \mathit{false} \\ \perp & \text{, } A_1(s,\pi,\alpha) = \perp \end{array} \right. \\ B(s,\pi,\beta) & = & \left\{ \begin{array}{ll} B_2(s,\pi,\beta) & \text{, } A_1(s,\pi,\beta) = \mathit{true} \\ B_3(s,\pi,\beta) & \text{, } A_1(s,\pi,\beta) = \mathit{false} \\ \perp & \text{, } A_1(s,\pi,\beta) = \perp \end{array} \right. \end{array}$$

Recursive definition.

$$\frac{\begin{array}{c}\forall 1 \leq i \leq n.\ E_i \triangleright V \cup \{x_1, \ldots, x_n\} \to \emptyset \\ E' \triangleright V \cup \{x_1, \ldots, x_n\} \to V' \\ E = \textbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \textbf{in}\ E'\end{array}}{E \triangleright V \to V'}$$

Let A_i be program evaluators for $E_i \triangleright V \cup \{x_1, \ldots, x_n\} \to \emptyset$ for $1 \leq i \leq n$ and A' and B' be evaluators for $E' \triangleright V \cup \{x_1, \ldots, x_n\} \to V'$. We define evaluators for $E \triangleright V \to V'$ as follows:

$$\begin{array}{l} A(s, \pi, \alpha) = \\ \quad \textbf{Letrec} \\ \quad\quad v_1 = A_1(s, \pi, \alpha_{|_V} \uplus [x_1/v_1, \ldots, x_n/v_n]), \\ \quad\quad \ldots \\ \quad\quad v_n = A_n(s, \pi, \alpha_{|_V} \uplus [x_1/v_1, \ldots, x_n/v_n]) \\ \quad \textbf{in}\ A'(s, \pi, \alpha \uplus [x_1/v_1, \ldots, x_n/v_n]) \end{array}$$

$$\begin{array}{l} B(s, \pi, \beta) = \\ \quad \textbf{Letrec} \\ \quad\quad v_1 = A_1(s, \pi, \beta \uplus [x_1/v_1, \ldots, x_n/v_n]), \\ \quad\quad \ldots \\ \quad\quad v_n = A_n(s, \pi, \beta \uplus [x_1/v_1, \ldots, x_n/v_n]) \\ \quad \textbf{in}\ B'(s, \pi, \beta \uplus [x_1/v_1, \ldots, x_n/v_n]) \end{array}$$

By Proposition 1, there exist algorithms computing the functions v_i for all $1 \leq i \leq n$.

Tuple constructions.

$$\frac{\forall 1 \leq i \leq n.\ E_i \triangleright V \to \emptyset \quad E = (E_1, \ldots, E_n)}{E \triangleright V \to \emptyset}$$

Let A_i be program evaluators for $E_i \triangleright V \to \emptyset$ for $1 \leq i \leq n$. We define the program evaluator A for $E \triangleright V \to \emptyset$ by:

$$A(s, \pi, \alpha) \quad = \quad \begin{cases} \bot & , \exists i : A_i(s, \pi, \alpha) = \bot \\ (A_1(s, \pi, \alpha), \ldots, A_n(s, \pi, \alpha)) & , \text{otherwise} \end{cases}$$

The query evaluator B can be derived from the program evaluator A since no variables are generated.

Tuple decompositions.

$$\frac{\begin{array}{c}E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \cup \{x_1, \ldots, x_n\} \to V' \\ E = \textbf{match}\ E_1\ \textbf{with}\ (x_1, \ldots, x_n)\ \textbf{in}\ E_2\end{array}}{E \triangleright V \to V'}$$

Let A_1 and be a program evaluator for $E_1 \triangleright V \to \emptyset$, and A_2 and B_2 evaluators for $E_2 \triangleright V \cup \{x_1, \ldots, x_n\} \to V'$. We define evaluators for $E \triangleright V \to V'$ as:

$$A(s,\pi,\alpha) = \begin{cases} A_2(s,\pi,\alpha \uplus [x_1/a_1,\ldots,x_n/a_n]) & , A_1(s,\pi,\alpha) \neq \bot \\ \quad \text{where } A_1(s,\pi,\alpha) = (a_1,\ldots,a_n) & \\ \bot & , \text{otherwise} \end{cases}$$

$$B(s,\pi,\beta) = \begin{cases} \{\beta'|_{V \cup V'} \mid \beta' \in B_2(s,\pi,\beta \uplus [x_1/a_1,\ldots,x_n/a_n]) & , A_1(s,\pi,\beta) \neq \bot \\ \qquad A_1(s,\pi,\beta) = (a_1,\ldots,a_n)\} & \\ \bot & , \text{otherwise} \end{cases}$$

Set constructions.

$$\frac{E \triangleright V \to \{x\}}{\{x \mid E\} \triangleright V \to \emptyset}$$

Let B_1 be a query evaluator for $E \triangleright V \to \{x\}$. We define the program evaluator A for $\{x \mid E\} \triangleright V \to \emptyset$ as follows:

$$A(s,\pi,\alpha) = \begin{cases} \bot & , \text{if } B_1(s,\pi,\alpha) = \bot \\ \{\beta'(x) \mid \beta' \in B_1(s,\pi,\alpha)\} & , \text{otherwise} \end{cases}$$

Since no variables are generated, the query evaluator B can be derived from the program evaluator A.

List constructions.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset}{E_1 :: E_2 \triangleright V \to \emptyset}$$

Let A_i be program evaluators for $E_i \triangleright V \to \emptyset$ for $i \in \{1,2\}$. We define the A program evaluator for $E_1 :: E_2 \triangleright V \to \emptyset$ as:

$$A(s,\pi,\alpha) = \begin{cases} \bot & , \exists i : A_i(s,\pi,\alpha) = \bot \\ A_1(s,\pi,\alpha) :: A_2(s,\pi,\alpha) & , \text{otherwise} \end{cases}$$

Since no variables are generated, the B query evaluator can be derived from the A program evaluator.

List decomposition.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \cup \{x,y\} \to V' \quad E_3 \triangleright V \to V' \quad E = \textbf{match } E_1 \textbf{ with } x :: y \textbf{ then } E_2 \textbf{ else } E_3}{E \triangleright V \to V'}$$

Let A_1 be a program evaluator for $E_1 \triangleright V \to \emptyset$, A_2 and B_2 evaluators for $E_2 \triangleright V \cup \{x,y\} \to V'$, and A_3 and B_3 evaluators for $E_3 \triangleright V \to V'$. We define evaluators for

$E \triangleright V \to V'$ as follows:

$$A(s,\pi,\alpha) = \begin{cases} A_2(s,\pi,\alpha \uplus [x/a, y/a']) & , A_1(s,\pi,\alpha) = a :: a' \\ A_3(s,\pi,\alpha) & , A_1(s,\pi,\alpha) = \mathit{nil} \\ \bot & , A_1(s,\pi,\alpha) = \bot \end{cases}$$

$$B(s,\pi,\beta) =$$

$$\begin{cases} \{\beta'_{|_{V \cup V'}} \mid \beta' \in B_2(s,\pi,\beta \uplus [x/a, y/a'])\} & , A_1(s,\pi,\beta) = a :: a' \\ B_3(s,\pi,\beta) & , A_1(s,\pi,\beta) = \mathit{nil} \\ \bot & , A_1(s,\pi,\beta) = \bot \end{cases}$$

Function application.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset}{E_1 E_2 \triangleright V \to \emptyset}$$

Let A_i be program evaluators for $E_i \triangleright V \to \emptyset$ for $i \in \{1,2\}$. We define the A program evaluator for $E_1 E_2 \triangleright V \to \emptyset$ as follows:

$$A(s,\pi,\alpha) = \begin{cases} f(A_2(s,\pi,\alpha)) & , f = A_1(s,\pi,\alpha) \neq \bot \\ \bot & , \text{otherwise} \end{cases}$$

Since no variables are generated, the B query evaluator can be derived from the program evaluator A.

Function definition.

$$\frac{E \triangleright V \cup \{x\} \to \emptyset}{\lambda x.E \triangleright V \to \emptyset}$$

Let A' be a program evaluator for $E \triangleright V \cup \{x\} \to \emptyset$. We define the A program evaluator for $\lambda x.E \triangleright V \to \emptyset$ as follows:

$$A(s,\pi,\alpha) = (\Lambda v \in [\![\textsc{Type}(x)]\!].A'(s,\pi,\alpha \uplus [x/v]))^{\bot}$$

Since no variables are generated, the B query evaluator can be derived from the A program evaluator.

Variable binding.

$$\frac{E \triangleright V \to \emptyset}{x = E \triangleright V \to \{x\}}$$

Let A' be a program evaluator for $E \triangleright V \to \emptyset$. We define the evaluators for $x = E \triangleright V \to \{x\}$ as follows:

$$A(s,\pi,\alpha) = \begin{cases} \bot & , \text{if } A'(s,\pi,\alpha_{|_V}) = \bot \\ \mathit{true} & , \text{if } \alpha(x) = A'(s,\pi,\alpha_{|_V}) \\ \mathit{false} & , \text{otherwise} \end{cases}$$

$$B(s,\pi,\beta) = \begin{cases} \bot & , \text{if } A'(s,\pi,\beta) = \bot \\ \{\beta \uplus [x/A'(s,\pi,\beta)]\} & , \text{otherwise} \end{cases}$$

Set Membership.

$$\frac{E \triangleright V \to \emptyset}{x \in E \triangleright V \to \{x\}}$$

Let A' be the program evaluator for $E \triangleright V \to \emptyset$. We define the evaluators for $x \in E \triangleright V \to \{x\}$ as follows:

$$A(s,\pi,\alpha) = \begin{cases} \bot & \text{, if } A'(s,\pi,\alpha_{|_V}) = \bot \\ \mathit{true} & \text{, if } \alpha(x) \in A'(s,\pi,\alpha_{|_V}) \\ \mathit{false} & \text{, otherwise} \end{cases}$$

$$B(s,\pi,\beta) = \begin{cases} \bot & \text{, if } A'(s,\pi,\beta) = \bot \\ \{\beta \uplus [x/v] \mid v \in A'(s,\pi,\beta)\} & \text{, otherwise} \end{cases}$$

□

10.2.2 Implementation

Large parts of the implementation of the in-memory evaluator are straightforward given by functional programs. What we have not yet explained is how functions will be represented. Functions may be contained in variable assignments as the input of the evaluator, but may also be produced as output.

In our implementation, we always represent functions by functional programs that compute the functions. For a function $\Lambda v \in \mathit{Int}.v + 3$, one can use the corresponding functional program $\mathbf{fun}(v)\ v + 3$ where $v \in \mathit{Int}$. For recursive functions $\mathbf{Letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \mathbf{in}\ E'$ we can assume functional programs for E_1, …, E_n and E and then construct the functional program for the letrec expression as in the proof of Proposition 1.

10.2.3 Examples

In the following we give examples for the evaluation of functions, recursive lets, logical formulas, and their combinations.

Functions When applied to the lambda expression $\lambda x.\ x + 5$ where $\textsc{Type}(x) = \mathbf{int}$ the implementation of the program evaluator A returns a functional program computing the total function $\Lambda i \in \mathit{Int}.\ i + 5$, which when applied to some integer $i \in \mathit{Int}$ returns $i + 5$. The implementation of the program evaluator A for the function application

$$(\lambda x : \mathbf{int}.\ x + 5)(3)$$

applies this functional program $\mathbf{fun}(v)\ v + 5$ where $v \in \mathit{Int}$, which when applied to argument 3 returns 8.

Recursion Let z be a variable of type $\textsc{Type}(z) = \mathbf{int}$. When applied to the recursive let expression

$$\mathbf{letrec}\ x_{fac} = \lambda z.\ \mathbf{if}\ z = 0\ \mathbf{then}\ 1\ \mathbf{else}\ z \cdot x_{fac}(z-1)\ \mathbf{in}\ x_{fac}$$

the implementation of the program evaluator A returns a functional program computing the faculty function:

$$fac = \mathbf{fun}(i)\ \mathbf{if}\ i = 0\ \mathbf{then}\ 1\ \mathbf{else}\ i \cdot fac\ (i-1)$$

where $i \in Int$. When applied to a nonnegative integer i this functional program computes $fac(i) = i!$, and otherwise returns $\perp$. This functional program can be obtained in a systematic manner from the letrec expression, as shown in the proof of Proposition 1.

Mixing Logics and Functions Let x be a variable of type $\textsc{Type}(x) = \mathbf{int}$. We consider the following logical formula which contains an application of a recursive function:

$$\exists x.\ x \in \{1,2,3\} \wedge fac(x) \geq 5$$

The query evaluator B for $x \in \{1,2,3\}$ will return the set of variable assignments $\{[x/1],[x/2],[x/3]\}$. From these, the query evaluator B of the conjunction will filter those assignments, for which program evaluator A of $fac(x) \geq 5$ returns true. This is the set $\{[x/3]\}$. Since this set is nonempty, the program evaluator A of the existential quantified formula will return *true*.

Chapter 11

Streaming Evaluation of λXP Queries

Contents

In this chapter we present a streaming algorithm that evaluates a network of early nested word automata, where navigational expressions are evaluated by eNWAs as described in Chapter 8.

11.1 Streaming Evaluators

In this section we describe how we evaluate λXP formulas over a stream. We are interested in two evaluators similar to Section 10.2: A program evaluator A_s that computes the semantics of a formula, and a query evaluator B_s that computes sets of extensions of variable assignment under which A_s becomes *true*. Both evaluators use registrations that store possible open descriptions of values and variable assignments, which can be completed when future values arrive on the stream.

11.1.1 Restrictions

For the streaming evaluation we need to impose the following restrictions on conditionals, list decompositions, and two new constants that we introduce. These restrictions are necessary to avoid exponential blow-ups and problems with program termination that arise by speculative execution that is required without the restrictions. For example, the test expression of a conditional may not yet be determined at some event of stream, since it requires more information from the stream for its evaluation. Therefore, one needs to speculatively guess that either the **then** or the **else** expression will be taken. A problem is the nontermination of an invalid branch that was a wrong guess. Furthermore, the choices for the guesses must be propagated throughout the network, leading to copies of the choices, which becomes expensive to resolve.

Furthermore, our streaming evaluators will maintain collections, i.e. set or lists, of nodes whose determination may require more information from the future of the stream. Such collections cannot always be decomposed by using list-matches, since list-matches will be strongly restricted to be acceptable for streaming.

Therefore, we add further constants to our language. In particular, we introduce the two mapping constants:

$$\begin{aligned} &\mathit{map\text{-}list}_{T,T'} : (T \to T') \times [T] \to [T'] \\ &\mathit{map\text{-}set}_{T,T'} : (T \to T') \times \{T\} \to \{T'\} \end{aligned}$$

which given some function f of type $T \to T'$ and a list or a set of type T, applies the function to every element of the list or set, to obtain a new list or set, respectively. This works fine, as long as termination is ensured for all applications of f to some value. We also add constant

$$\mathit{string\_value} : \mathbf{node} \to \mathbf{string}$$

to our language, that can be defined as in Section 7.2.3, but as a constant it can be implemented in any preferable way. It computes the string value of a given node. Then, for example, *map-list string_value* can convert a list of nodes to the list of their string values, even though the membership of some node to the input list will be decided only lately. In this case, one will apply the function to the candidate nodes in a speculative manner. This is safe, as long as speculative function applications will not lead to nontermination.

Definition 19. *We call a* λXP *expression* E stream-acceptable *if* E *is well-typed and satisfies the following two conditions:*

(1) for subexpression of E *that is either a conditional*

$$\mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2$$

or a branching match-expression

$$\mathbf{match}\ E_0\ \mathbf{with}\ x :: y\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2$$

with types $E_i : T_i$ *for* $1 \leq i \leq 3$*, either of the following holds:*

(a) the types T_1 and T_2 do not contain the type **node** *as a subterm and the expressions E_1 and E_2 do not contain any axis and any constant whose semantics depends on nodes, or*

(b) the type T_0 does not contain the type **node** *as a subterm and the expression E_0 does not contain any axis and any constants whose semantics depends on nodes.*

(2) all occurrences of the constants map-list or map-set in E must be in applications of the form

$$\textit{map-list } E' \qquad \textit{or} \qquad \textit{map-set } E'$$

where E' is some expression that is terminating in that it does neither contain ***letrec*** *nor any free variables or constants, with the exception of*

$$\textit{string\_value}$$

(or other constants that may not lead to nontermination).

Restriction (1a) allows for a late evaluation of branches E_1 and E_2 of conditionals or branching match expressions. Since neither T_1 and T_2 do contain the type **node** as a subterm and neither E_1 nor E_2 contain any axes, the expressions E_1 and E_2 do not depend on nodes on the stream, so that can be evaluated at any time point, without any needs of stream buffering. For example, consider the expression

$$\textbf{if } \textit{ch}^*(a) \textbf{ then } \textit{fac}(\textit{string\_value}(\textit{1st-node}())) \textbf{ else } \textit{fac}(\textit{string\_value}(\textit{2nd-node}()))$$

that is run on a document which has no a-labeled nodes, while the documents first node's string value is -1 and the second node's string value is 3. *1st-node* and *2nd-node* are constants that return the first and second node of the root, respectively:

$$\textit{1st-node} : \textbf{unit} \to \textbf{node} \qquad \text{and} \qquad \textit{2nd-node} : \textbf{unit} \to \textbf{node}$$

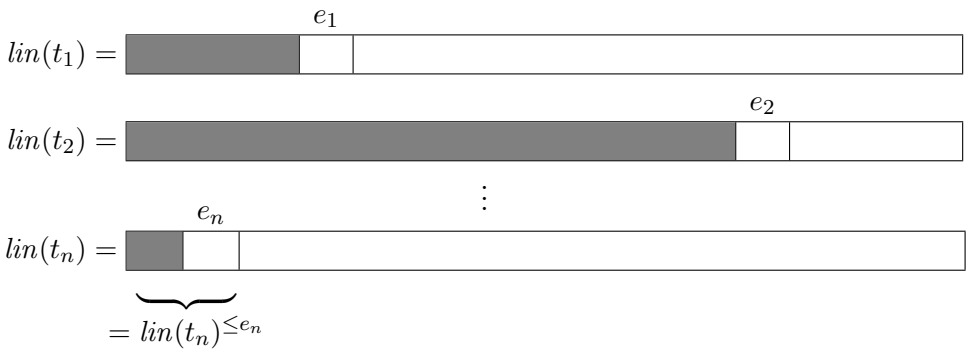
and *fac* is the faculty function defined in Section 6.2.7 that is defined for positive integers only. The correct evaluation of the expression on the stream which contains no a-labeled node should terminate with 6, since the evaluation of the test expression of the conditional selects the **else** branch, but however, before arriving at the end of the stream to determine whether an a-labeled descendant exists, when doing all computation speculatively, one would be forced to also evaluate $\textit{fac}(-1)$, which would lead to nontermination.

With restriction (1b) the test expression of the conditional and branching match expressions will be determined right at the node when the branches are started, since independent of future nodes on the stream. Then one chooses the respective branch depending on the evaluation of the test expression and whether the list is empty or not, respectively.

Mapping expressions *map-list* E and *map-set* E allow to decompose lists and sets containing elements whose membership to the list or set may depend on information from the future of stream. However, we must ensure that the function application of the function by E on elements of the collection does not run into nontermination. Restriction (2) thereby forbids **letrec** expressions, free variables, and other constants, which may lead to nontermination.

11.1.2 Linearizing a Sequence of Trees

Let s be a sequence of trees $s = t_1, \ldots, t_n$. A streaming evaluator for s consumes the events of linearizations $lin(t_1), \ldots, lin(t_n)$. At every time point, i.e. after having consumed already all events of linearizations $lin(t_i)$ up to and excluding an event e_i for all $1 \leq i \leq n$, then the evaluator has the choice to read next any of the events e_i for some $1 \leq i \leq n$:



By $l^{\leq e}$ we denote the prefix of a linearization l that ends with event e. A prefix of a linearization of s is thus a tuple $(lin(t_1)^{\leq e_1}, \ldots, lin(t_n)^{\leq e_n})$, which we denote by $lin(s)^{\leq \vec{e}}$ where $\vec{e} = (e_1, \ldots, e_n)$. Notice that there are many possible linearizations of a sequence of trees: one for every different shuffle of events of the linearizations of the trees.

11.1.3 Updates

In streaming, values are computed over the stream incrementally, such that one needs a facility to modify them in an incremental manner. We define the modifications via update commands *Upds* listed in Figure 11.1 whose types $\mathbf{upds}(T)$ are defined in Figure 11.2. When no update is required, one can make use of the command *none*. Update commands can be composed via "&". Atomic future values • can be instantiated with a value v via command $instantiate(v)$. Tuples are updated with an tuple of updates. Sets and lists can be extended and their elements can be updated and resolved, such that the element remains in the collection or is removed. Finally, variable assignments can be extended with a new record and values of variables may be updated.

In Figure 11.3 we present the *apply* function that given some value v and an update command c returns the by c updated value of v.

$c ::=$	none		no update
$\mid$	$c_1 \& c_2$		two consecutive updates
$\mid$	$\mathit{instantiate}(v)$	$v : T \in \mathcal{T}_{atom}$	instantiating $\bullet$ by v
$\mid$	$(c_1, \ldots, c_n)$		updating tuples
$\mid$	$\mathit{extend\text{-}set}(v)$	$v : \{T\}$	extending sets by set v
$\mid$	$\mathit{upd\text{-}elem\text{-}set}(v, c)$	$v : T$	updating element v of a set
$\mid$	$\mathit{resolve\text{-}elem\text{-}set}(e^?, b)$	$e^? : T, b : \mathbf{bool}$	resolving element $e^?$ by b
$\mid$	$\mathit{extend\text{-}list}(v)$	$v : [T]$	extending lists by list v
$\mid$	$\mathit{upd\text{-}elem\text{-}list}(i, c)$	$i : \mathbf{int}$	updating elements at index i in a list
$\mid$	$\mathit{resolve\text{-}elem\text{-}list}(i, b)$	$i : \mathbf{int}, b : \mathbf{bool}$	resolving element $e^?$ at position i by boolean b
$\mid$	$\mathit{extend\text{-}var\text{-}ass}(x, v)$	$x : T, v : T$	extending variable assignment by $[x/v]$
$\mid$	$\mathit{upd\text{-}var\text{-}ass}(x, v, c)$	$x : T, v : T$	updating assignment x to v by c

Figure 11.1: Update commands c.

$$\frac{}{\mathit{none} : \mathbf{upds}(T)}\quad\frac{c_1 : \mathbf{upds}(T) \quad c_2 : \mathbf{upds}(T)}{c_1 \& c_2 : \mathbf{upds}(T)}$$

$$\frac{T \in \mathcal{T}_{atom} \quad v \in T}{\mathit{instantiate}(v) : \mathbf{upds}(T)}\quad\frac{c_1 : \mathbf{upds}(T_1) \quad \ldots \quad c_n : \mathbf{upds}(T_n)}{(c_1, \ldots, c_n) : \mathbf{upds}(T_1 \times \ldots \times T_n)}$$

$$\frac{v : \{T\}}{\mathit{extend\text{-}set}(v) : \mathbf{upds}(\{T\})}\quad\frac{v : T \quad c : \mathbf{upds}(T)}{\mathit{upd\text{-}elem\text{-}set}(v, c) : \mathbf{upds}(\{T\})}$$

$$\frac{e^? : T \quad b : \mathbf{bool}}{\mathit{resolve\text{-}elem\text{-}set}(e^?, b) : \mathbf{upds}(\{T\})}$$

$$\frac{v : [T]}{\mathit{extend\text{-}list}(v) : \mathbf{upds}([T])}\quad\frac{i : \mathbf{int} \quad c : \mathbf{upds}(T)}{\mathit{upd\text{-}elem\text{-}list}(i, c) : \mathbf{upds}([T])}$$

$$\frac{i : \mathbf{int} \quad b : \mathbf{bool}}{\mathit{resolve\text{-}elem\text{-}list}(i, b) : \mathbf{upds}([T])}$$

$$\frac{x : T \quad x \in V \quad v : T}{\mathit{extend\text{-}var\text{-}ass}(x, v) : \mathbf{upds}(\mathbf{ass}_V)}\quad\frac{x : T \quad x \in V \quad v : T \quad c : \mathbf{upds}(T)}{\mathit{extend\text{-}var\text{-}ass}(x, v) : \mathbf{upds}(\mathbf{ass}_V)}$$

Figure 11.2: Update commands and their types $\mathbf{upds}(T)$.

```
letrec
  apply = fun(v,c)
    case c
    of none then none
    of c1&c2 then apply(apply(v,c1),c2)
    of instantiate(v') then if v=• then v' end
    of (c1,...,cn) then
      case v of (v1,...,vn) then (apply(v1,c1),...,apply(vn,cn)) end
    of extend-set(v') then v ∪ v'
    of upd-elem-set(e,c') then v \ {e} ∪ {apply(e,c')}
    of resolve-elem-set(e?,b) then if b then v \ {e?} ∪ {e} else v \ {e?}
    of extend-list(v') then listconcat(v,v')
    of upd-elem-list(i,c') then list-apply-pos(v,i,λx.apply(x,c'))
    of resolve-elem-list(i,b) then if b then case v[i] of e? then
        list-set(v,i,e) end else list-remove(v,i)
    of extend-var-ass(x,v') then v ⊎ [x/v']
    of upd-var-ass(x,v',c') then v_|v.dom()\{x}} ⊎ [x/apply(v',c')]
  end
in
  apply
end
```

Figure 11.3: Function to apply update commands.

11.1.4 Extended Types

We extend the types of λXP by types $\mathbf{ass}_V$ for variable assignments, type $\mathbf{proc}(T)$ of procedures that are like functions of type $T \to \mathbf{unit}$ except that they do not produce any output, type $\mathbf{upds}(T)$ for update commands, variable types $\{x\}$, and event types **event**:

$$\begin{array}{rcl} T & ::= & A \mid T_1 \times \ldots \times T_n \mid [T] \mid \{T\} \mid T_1 \cup T_2 \mid T_1 \to T_2 \\ & \mid & \mathbf{ass}_V \mid \mathbf{proc}(T) \mid \mathbf{upds}(T) \mid \{x\} \mid \mathbf{event} \end{array}$$

11.1.5 Open Values

For the evaluation of λXP formulas we have to compute s-values incrementally, since the stream is consumed event per event. Given a prefix $lin(s)^{\leq \vec{e}}$ of a linearization of s, we can thus only compute $lin(s)^{\leq \vec{e}}$-values, that are open values that we define in the following.

Open values are values such as atomic values, tuples, and functions, but where sets are replaced by open sets, lists by open lists, and where the • value is added. The open • value represents a future value that can only be determined once more information from the stream is known. For example, it may be the future value of a filter that requires to read more from the stream for the evaluation to *true* or *false*. Sets and lists may also contain uncertain elements $v^?$, which are atomic values whose selection depends on the future of the stream. It may be that not all the elements of a set are yet known, for which we consider open sets $S \cup \bullet$ where

$$
\begin{array}{lcl}
[\![T]\!]_s^{\leq \vec{e}} & = & [\![T]\!]_s \setminus \{\pi \mid \pi \notin \text{nodes}(lin(s)^{\leq \vec{e}})\} \cup \{\bullet\} \qquad T \in \mathcal{T}_{atom} \\
[\![T_1 \times \ldots \times T_n]\!]_s^{\leq \vec{e}} & = & [\![T_1]\!]_s^{\leq \vec{e}} \times \ldots \times [\![T_n]\!]_s^{\leq \vec{e}} \\
[\![\{T\}]\!]_s^{\leq \vec{e}} & = & \{\{v_1, \ldots, v_n\} \cup \bullet \mid v_i = v_i' \text{ or } v_i = v_i'^{?}, v_i' \in [\![T]\!]_s^{\leq \vec{e}}\} \\
[\![[T]]\!]_s^{\leq \vec{e}} & = & \{v_1 :: \ldots :: v_n :: \bullet \mid v_i = v_i' \text{ or } v_i = v_i'^{?}, v_i' \in [\![T]\!]_s^{\leq \vec{e}}\} \\
[\![T_1 \cup T_2]\!]_s^{\leq \vec{e}} & = & [\![T_1]\!]_s^{\leq \vec{e}} \cup [\![T_2]\!]_s^{\leq \vec{e}} \\
[\![T_1 \to T_2]\!]_s^{\leq \vec{e}} & = & [\![T_1]\!]_s^{\leq \vec{e}} \to ([\![T_2]\!]_s^{\leq \vec{e}} \cup \{\bot\}) \\
[\![\mathbf{ass}_V]\!]_s^{\leq \vec{e}} & = & \text{assignments of variables } x \in V \text{ of some type } T \\
 & & \text{to open values in } [\![T]\!]_s^{\leq \vec{e}} \\
[\![\mathbf{proc}(T)]\!]_s^{\leq \vec{e}} & = & [\![T]\!]_s^{\leq \vec{e}} \to () \\
[\![\mathbf{upds}(T)]\!]_s^{\leq \vec{e}} & = & \text{as defined in Figure 11.2} \\
[\![\{x\}]\!]_s^{\leq \vec{e}} & = & \{x\} \\
[\![\mathbf{event}]\!]_s^{\leq \vec{e}} & = & events(s)
\end{array}
$$

Figure 11.4: Open values $[\![T]\!]_s^{\leq \vec{e}}$, where T is a type, s a sequence of trees, and $lin(s)^{\leq \vec{e}}$ is a prefix of $lin(s)$ that ends with events $\vec{e}$.

only the elements of S are known so far. Similarly open lists can be open to the right, such that the last element in the list is $\bullet$, for example open list $1 :: 2 :: 3 :: \bullet$. More precisely, we define *open values* $[\![T]\!]_s^{\leq \vec{e}}$ in Figure 11.4 for any extended type T and prefix $lin(s)^{\leq \vec{e}}$ of a linearization of a sequence s of data trees.

For atomic types $T \in \mathcal{T}_{atom}$ open values must belong to $lin(s)^{\leq \vec{e}}$-values and thus exclude any node that comes in the future of the stream. The selection of uncertain atomic open values $v^?$ depends on the future of the stream. They may only appear within sets and lists, and may thus be deleted later on. Open tuples are tuples that contain open values. Open sets are sets that may grow with the stream and contain uncertain elements. Open lists are lists $v_1 :: \ldots :: v_n :: \bullet$ with an open end such that it may still grow with the stream containing possibly uncertain elements. Open functions are functions whose arguments and values are open values.

11.1.6 Registrations

We are interested in a program evaluator that computes the semantics of a formula incrementally while consuming the stream, and in a query evaluator that computes incrementally extensions of variable assignments under which some formula becomes *true*. This requires registrations that store in tables tuples of open values and variable assignments to open variable.

Definition 20. *Let V be a finite subset of typed variables, and let T be some type. A* registration *with these parameters is a tuple*

$$R = (R.dom, R.set\text{-}val, R.add\text{-}inst, R.remove, R.upd\text{-}ass)$$

of programs, that operate on a finite table as illustrate in Figure 11.5. This table maps entries of type $T' = \mathbf{node} \times \mathbf{ass}_V$ to values of type T in a functional manner. The programs are doing the following:

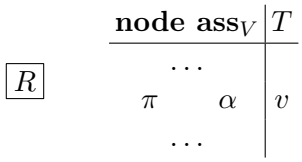


Figure 11.5: A registration table.

- $R.dom : \mathbf{unit} \to T'$ *is always applied to the empty tuple. It returns the current set of entries of type* T' *to which the registration's table assigns values.*
- $R.add\text{-}inst : \boldsymbol{proc}(T')$ *can be applied to any entry of type* T' *that is new to the table, and then adds it to the table with value* $\bullet$*.*
- $R.set\text{-}val : \boldsymbol{proc}(T' \times T)$ *is given as arguments an entry of type* T' *in the table and a value of type* T*. It then adds the entry with this value to the table.*
- $R.get\text{-}val : T' \to T$ *is a partial function that can be applied to an entry of type* T'*. It then returns the unique value of type* T *that the table assigns to this entry, while assuming that it is there.*
- $R.remove : \boldsymbol{proc}(T')$ *receives an entry of type* T' *as argument and removes it from the table, while assuming that is was there.*
- $R.upd\text{-}ass : \boldsymbol{proc}(T' \times \mathbf{ass}_V)$ *receives an entry of type* T' *and a variable assignment of type* $\mathbf{ass}_V$ *and it assumes that the entry exists in the table. It then replaces the variable assignment of the entry with the given variable assignment, preserving the entrie's value.*

Furthermore, we introduce *master-registrations* that extend registrations with further programs by which to govern a tuple of slave registrations by providing additional functionality.

Definition 21. *Let* n *be a natural number,* R *is a registration of type* T *with entries of type* $T' = \mathbf{node} \times \mathbf{ass}_V$ *for some set of typed variables* V*, and* R_i *are subregistrations of type* T_i *for all* $1 \leq i \leq n$*. A* master-registration *is a tuple of programs:*

$$\mathcal{M}(R, R_1, \ldots, R_n) = (\mathcal{M}.init\text{-}collection, \mathcal{M}.collect, \mathcal{M}.get\text{-}collection)$$

These programs are doing the following:

- $\mathcal{M}.init\text{-}collection : \boldsymbol{proc}(T' \times \mathbf{int})$ *sets for some entry of type* T' *and some integer* $1 \leq i \leq n$ *the update command for subregistration* R_i *to none,*
- $\mathcal{M}.collect : \boldsymbol{proc}(\bigcup_{1 \leq i \leq n} T' \times \mathbf{upds}(T_i) \times \mathbf{int})$ *collects update commands for some entry of type* T'*, some update command* c*, and some integer* $1 \leq i \leq n$*. Update command* c *is collected, by replacing the already collected update command* c' *of registration* R_i *by* $c'\&c$*, and*

- $\mathcal{M}.\mathit{get\text{-}collection} : \bigcup_{1 \le i \le n} T' \times \mathbf{int} \to \mathbf{upds}(T_i)$ *returns for some entry of type* T' *and some integer* $1 \le i \le n$ *the collected update command for registration* R_i.

11.1.7 Program and Query Evaluators

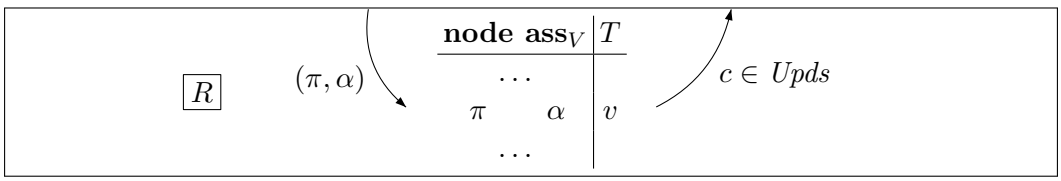
In streaming we define program and query evaluators similar to the in-memory case, which however compute semantics of formulas and variable assignments incrementally. Let E be a λXP formula of type T with mode $E \triangleright V_1 \to V_2$.

Program Evaluators. Program evaluators A_s compute a semantic description of type T of E over some linearization $\mathit{lin}(s)$ for some open node $\pi : \mathbf{node}$ and some open variable assignment $\mu : \mathbf{ass}_{V_1 \cup V_2}$. Evaluator A_s maintains a registration R with typed variables $V_1 \cup V_2$ and result type T. Program evaluators $A_s = (\mathit{get\text{-}val}, \mathit{is\text{-}done}, \mathit{remove}, \mathit{add\text{-}inst}, \mathit{upd\text{-}val})$ are defined via the following functions, where $T' = \mathbf{node} \times \mathbf{ass}_{V_1 \cup V_2}$:

- $\mathit{get\text{-}val} : T' \to T$ is a partial function,
- $\mathit{is\text{-}done} : T' \to \mathbf{bool}$ is a partial function,
- $\mathit{remove} : \mathbf{proc}(T')$,
- $\mathit{add\text{-}inst} : T' \to \mathbf{upds}(T)$, and
- $\mathit{upd\text{-}val} : \bigcup_{x \in V_1 \cup V_2} \mathbf{event} \times 2^{T' \times \{x\} \times \mathbf{upds}(\textsc{Type}(x))} \to 2^{T' \times T \times \mathbf{upds}(T)}$.

Function $\mathit{get\text{-}val}(\pi, \mu)$ returns the value stored in R for π and μ by calling $R.\mathit{get\text{-}val}(\pi, \mu)$. Function $\mathit{is\text{-}done}(\pi, \mu)$ returns whether the open value computed by A_s is determined. Procedure $\mathit{remove}(\pi, \mu)$ removes the entry with respect to π and μ and thereby also calls $R.\mathit{remove}(\pi, \mu)$. Function $\mathit{add\text{-}inst}(\pi, \mu)$ created a new entry in R for the computation of the semantic of E, but for a new node π and variable assignment μ. The function calls $R.\mathit{add\text{-}inst}(\pi, \mu)$ and returns the update command, by which the initial $\bullet$-value in R for π and μ was updated. Lastly, function $\mathit{upd\text{-}val}(e, \mathit{AssUps})$ updates entries in R with respect to event e and $(\pi, \mu, x, c) \in \mathit{AssUps}$, where the latter requires to update entry $(\pi, \mu) \in R.\mathit{dom}()$ by applying c to the value of $x \in \mu.\mathit{dom}()$. Not all entries may evolve at the new event e however. The update function therefore returns the set of all tuples (π, μ, v, c), such that the value v of entry (π, μ) was updated with c to some new value v'.

The evaluator is executed in an *environment* which feeds R with instructions to create (*add-inst*), evolve (*upd-val*), and remove (*remove*) entries:



The incoming edge with label (π, α) represents that the environment has called *add-inst*(π, α) that created entry (π, α) in R. The outgoing edge indicates that a call to *upd-val*$(e, AssUps)$ at some event e with variable assignment update $AssUps$ updated the previous value of the entry with update command $c \in Upds$ to value v. Notice, that the program and query evaluators may compute values for some future node $\bullet$. In streaming we require such pre-computations for nodes that have not yet been seen on the stream, since the evaluation at nodes may require information from the past of the stream (via backward axes), while a streaming evaluator cannot go back on the stream. However given an environment such pre-computations are thereby not arbitrary. For example, when interested in computing the node labels for ancestor nodes, then the environment is in charge of running the a program evaluator A_s for query $lab_{name}(y)$ only for ancestor nodes y. The environment in this case is then another evaluator for the selection of ancestor nodes, that pre-computes nodes that may be ancestors to future nodes and thus runs A_s for these precomputed nodes.

Query Evaluators. Query evaluators B_s compute extensions of variable assignments under which formula E becomes *true*. Given some open node $\pi : \mathbf{node}$ and some open variable assignment $\mu : \mathbf{ass}_{V_1}$ it means to generate all possible extensions $\mu' : \mathbf{ass}_{V_1 \cup V_2}$ of μ. Evaluator B_s maintains a registration R with typed variables V_1 and result type $2^{\mathbf{ass}_{V_1 \cup V_2}}$. Program evaluators $B_s = (\textit{get-val}, \textit{is-done}, \textit{remove}, \textit{add-inst}, \textit{upd-val})$ are defined via the following functions, where $T' = \mathbf{node} \times \mathbf{ass}_{V_1}$ and $T'' = 2^{\mathbf{ass}_{V_1 \cup V_2}}$:

- $\textit{get-val} : T' \to T''$ is a partial function,
- $\textit{is-done} : T' \to \mathbf{bool}$ is a partial function,
- $\textit{remove} : \mathbf{proc}(T')$,
- $\textit{add-inst} : T' \to \mathbf{upds}(T'')$, and
- $\textit{upd-val} : \bigcup_{x \in V_1} \mathbf{event} \times 2^{T' \times \{x\} \times \mathbf{upds}(\textsc{Type}(x))} \to 2^{T' \times T'' \times \mathbf{upds}(T'')}$.

similar to program evaluators, but with different signatures.

11.1.8 Running Evaluators over Streams

Let $s = t_1, \ldots, t_n$ be a stream of data trees $t_1, \ldots, t_n$. Let E be a well-typed and well-moded λXP formula of type T and mode $E \triangleright V_1 \to V_2$. Let $\mu : \mathbf{ass}_{V_1 \cup V_2}$ be an assignment of input variables in $V_1 \cup V_2$, let $\pi \in \{\textit{root}(t_i) \mid 1 \leq i \leq n\}$, and let A_s be a program evaluator for $E \triangleright V_1 \to V_2$. The program evaluator A_s is run over stream s according to the evaluation function *eval* of Figure 11.6. There, *list-of-events*(s) computes a *stream-merge* of linearizations $\textit{lin}(t_i)$ of s returning a list of events of $\textit{lin}(s)$ in an implementation dependent order. Note that at any time point, open values $[\![T]\!]_s^{\leq \vec{e}}$ are stored in the registration for A_s.

```
//Let A_s be a program evaluator for E ▷ V_1 → V_2

eval_s =
  fun (π, μ)
      (get-val, is-done, remove, add-inst, upd-val) = A_s
      {x_1, ..., x_n} = V_1 ∪ V_2
      add-inst(•, μ)
  in
      letrec
       eval' =
        fun (l)
          case l
          of e :: l' then
              if e = (op, π) then add-inst(π, μ) else skip end
              upd-val(e, ∅)
           in
              if is-done(π, μ) then
                 get-val(π, μ)
              else
                 eval'(l')
              end
          of nil then
             get-val(π, μ)
          end
        end
      in
        eval'(list-of-events(s))
  end
```

Figure 11.6: Evaluation of program evaluators over streams s.

```
// for any link function on a registration R with variables W
//     link : node × ass_W → ass_W' for some W'
// generate a function computing inverse links to entries of R:
//     gen-inv-links(R, link) : node × ass_W' → 2^(node×ass_W)

gen-inv-links(R, link) =
    fun (π : node, γ' : ass_W')
        {(π, γ) ∈ R.dom() | link(π, γ) = γ'}
    end
```

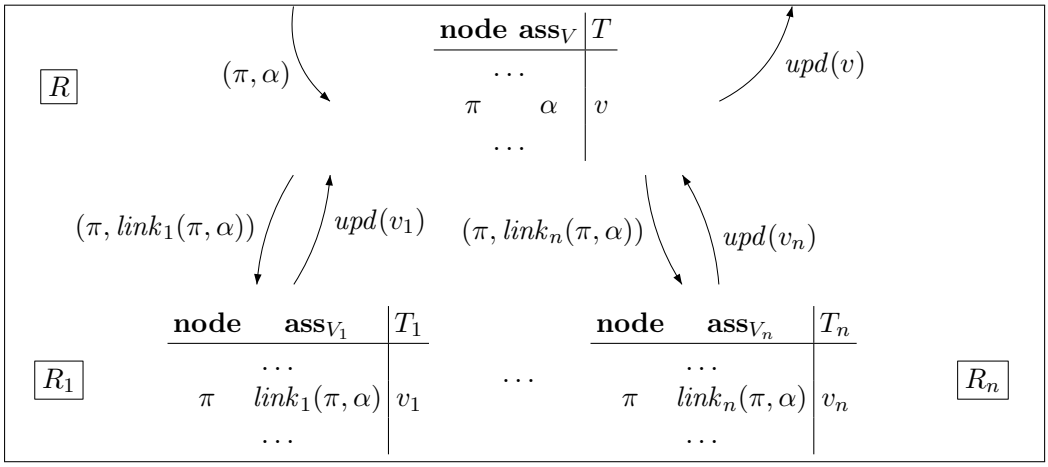
Figure 11.7: Generator of inverse links function.

11.2 Evaluators

In this section we present the streaming algorithms that evaluate a network of simple well-typed and well-moded λXP formulas with eliminated backward axes. For this we need to define program and query evaluators for all λXP formulas. Before defining them inductively with respect to the mode system, we will present and explain commonly used generic functions that will be used by our evaluators.

11.2.1 Generic Functions

Since the evaluators are constructed inductively, the registrations have the following form:



There, some evaluator A_s updates a registration R with variables in V, while subevaluators A_{is} will update registrations R_i for $1 \leq i \leq n$. Evaluator A_s maintains link relations $link_i : \mathbf{node} \times \mathbf{ass}_V \to \mathbf{ass}_{V_i}$ that uniquely link an entry $(\pi, \alpha) \in R.dom()$ to entries $(\pi, link_i(\pi, \alpha)) \in R_i.dom()$. However, an entry in a subregistration may be linked to various entries in the parent registration. In Figure 11.7 we present a generator for inverse link function that map to sets of pairs of nodes and variable assignments in the parent registration.

```
// for any link function on a registration
// generate a function gen-ass-com that computes update commands

gen-ass-com(link) =
  fun (AssUps, U)
    AssUps' = set.new()
    for (π, γ, x, c) ∈ AssUps do
      AssUps'.add(π, link(π, γ), x, c)
    end
   in
    AssUps'.content()
  end
```

Figure 11.8: Generator of functions that update assignment commands.

Registration then exchange update commands that are communicated among linked entries. Given updates c_i of values of entries in R_i one can then compute update commands for entries in the parent registration R to which is linked with respect to the inverse link function. Update commands are thus computed in a bottom up fashion. However, one must also pass updates of variable assignments top down. This becomes evident, for example, when decomposing tuples. Consider mode rule

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \cup \{x_1, \ldots, x_n\} \to V' \quad E = \textbf{match } E_1 \textbf{ with } (x_1, \ldots, x_n) \textbf{ in } E_2}{E \triangleright V \to V'}$$

When subexpression E_1 evaluates to some n-tuple, then E_2 is evaluated while assigning to variables x_i the value of the i-th component of the n-tuple. In streaming, the values of the components of the open tuple may not yet be determined. Therefore, whenever some component of the tuple is updated, one also has to update the corresponding variable assignment that E_2 contains. In Figure 11.8 we define generator *gen-ass-com* that given some link definition returns a function that starting from some update for variable assignments *AssUps*, returns a new update for variable assignments to be passed down to subevaluators according to the link definition.

In the beginning before evaluating over the stream we must initialize all registrations with an initial entry of the future node $\bullet$ and some initial variable assignment where every variable is mapped to $\bullet$. This is necessary for the automata registrations that precompute backward axes nodes. Besides the initial entries one can create new entries in some registration R via function *add-inst* of R. Due to our restrictions it holds the following invariant: When calling *add-inst*(π, μ) at some event of the stream, either the event is the opening event of node π, or R answers an expression whose type contains not the type **node** as subterm. The function can be defined for most λXP constructs by the functions generator of Figure 11.9 that is parameterized by *add-inst*$_i$ and *link*$_i$ functions of subregistrations and function *get-upd-com* that computes update commands. The generated function obtains the

```
// R is registration with variables W and type T with streams s
// that has subregistrations R_i with the same streams s. For i:
// - function add-inst_i instantiates entries of registration R_i
// - function link_i converts entries of R to entries of R_i
// - function get-upd-com returns update commands of R based on the
    update commands of R_i

gen-add-inst(R, add-inst_1, ..., add-inst_n, link_1, ..., link_n, get-upd-com) =
    fun (π : node, γ : ass_W)
        R.add-inst(π, γ)
        c_1 = add-inst_1(π, link_1(π, γ))
        ...
        c_n = add-inst_n(π, link_n(π, γ))
        c = get-upd-com(π, γ, c_1, ..., c_n))
        R.set-val(π, γ, apply(R(π, γ), c))
      in
        c
    end
```

Figure 11.9: Generator of functions that instantiate registration entries.

```
// R is registration with variables W and subregistration R_i on
// the same streams. For all i:
// - procedure remove_i removes entries of registration R_i
// - function link_i convert entries of R to entries of R_i

gen-remove(R, remove_1, ..., remove_n, link_1, ..., link_n) =
  proc (π : node, γ : ass_W)
      remove_1(π, link_1(π, γ))
      ...
      remove_n(π, link_n(π, γ))
      R.remove(π, γ)
  end
```

Figure 11.10: Generator of remove procedures.

updates c_i by the *add-inst*$_i$ function applied to entries that are in *link*$_i$ relation to the input node π and variable assignment γ, then it computes the update commands c for R by calling *get-upd-com* on the basis of all c_i. Finally, it sets the value of R for π and γ, applying the result update command c to the initial $\bullet$-value $R(\pi, \gamma)$.

Entries in R may also be removed, which can be defined for most evaluators by procedures of the generator in Figure 11.10. Via the *link*$_i$ functions and *remove*$_i$ procedure for subregistrations, the remove procedures by the generator removes entries in subregistrations to which entry $(\pi, \gamma) \in R.dom()$ is linked, and it removes entry (π, γ) itself.

Last but not least, in Figure 11.11 we present a generator for update functions that implements a rewriting of entries in R that returns a set of tuples (π, γ, v, c) of entries $(\pi, \gamma) \in R.dom()$ whose value v was updated by c at some event e of

some stream s. First, one updates the variable assignments in R with respect to tuples $(\pi, \gamma, x, c) \in \mathit{AssUps}$: An entry with node π and variable assignment γ is updated with the value that is obtained by applying command c to the value of x in γ. Next, one updates entries of subregistrations R_i by calling the respective *upd-val*$_i$ functions, while an update U_i may cause a change in a variable assignment of a sibling registration, for which *get-ass-com* is called. Notice that answer tuples in U_i concern only those entries in R_i whose value changed at event e. Similarly for R one wants to return only those answers of entries which value has changed. Therefore, we initialize a new collection, and collect all tuples (π, γ, c, i) where a linked entry of subregistration R_i was updated with command c. Then, for all entries in R for which some linked entry of a subregistrations was updated, one calls the *get-upd-com* function for R to compute the updates for the respective entry. We note that $\mathcal{M}.\mathit{get\text{-}collection}(\pi, \gamma, i)$ is equal to update command *none* if the linked entry in R_i was not updated. Finally, we set the new value in R, applying the newly computed update command to the old value in R, and we can safely accumulate tuples in the answer set.

11.2.2 Inductive Construction of Evaluators

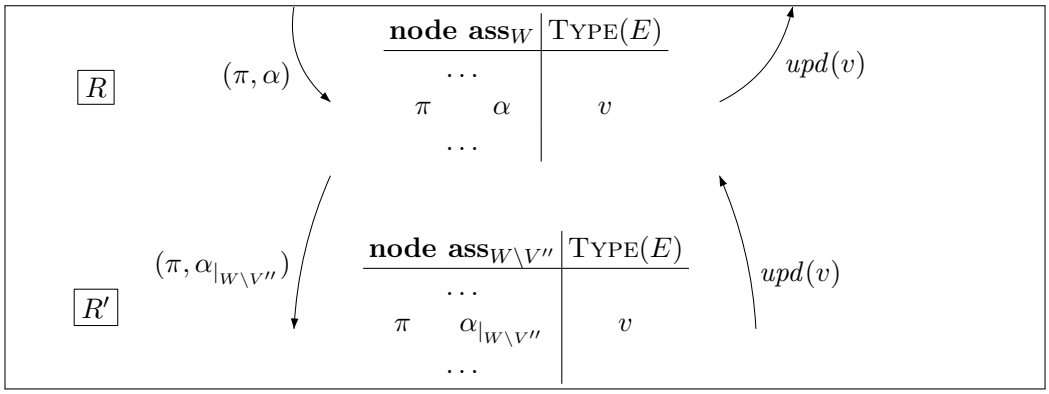
The construction of A_s and B_s is by simultaneous induction on derivations of judgements $E \triangleright V_1 \to V_2$.

Induction:

Submode rule 1. Suppose that the judgement $E \triangleright V_1 \to V_2$ is derived by the first rule for submodes:

$$\frac{E \triangleright V \to V'}{E \triangleright V \cup V'' \to V'}$$

Let A'_s be a program evaluator evaluating $E \triangleright V \to V'$ while assuming values for all variables in V'. We have to define a program evaluator A_s that evaluates $E \triangleright V \cup V'' \to V'$ given values for V'. The idea is as follows. Program evaluator A_s updates a registration R with variables in $W = V \cup V' \cup V''$, while program evaluator A'_s will update a registration R' with variables in $W \setminus V''$ that contains the corresponding entries projected to $W \setminus V''$.



```
// R is registration with variables W and type T with streams s
// that has subregistration R_i with the same streams s. For all i:
// - function upd-val_i updates entries of registration R_i
// - function link_i converts entries of R to entries of R_i
// - function get-upd-com returns update commands of R based on the
    updated commands of R_i
// - function get-ass-com_i computes assignment updates

gen-upd-val(R, upd-val_1, ..., upd-val_n, link_1, ..., link_n, get-upd-com, get-ass-com_1, ...,
 get-ass-com_n) =
   let
      inv-links_1 = gen-inv-links(R, link_1)
      ...
      inv-links_n = gen-inv-links(R, link_n)
   in
      fun(e ∈ events(s), AssUps)
        for (π, γ, x, c) ∈ AssUps do R.upd-ass(π, γ, γ|_{W\{x}} ⊎ [x/apply(γ(x), c)]) end
        U_1 = upd-val_1(e, get-ass-com_1(AssUps, ∅))
        U_2 = upd-val_2(e, get-ass-com_2(AssUps, U_1))
        ...
        U_n = upd-val_n(e, get-ass-com_n(AssUps, U_{n-1}))
        M = master-registration.new(R, R_1, ..., R_n)
        for (π, γ) ∈ R.dom() do
          for 1 ≤ i ≤ n do M.init-collection(π, γ, i) end
        end
        for 1 ≤ i ≤ n do
          for (π, γ', v, c) ∈ U_i do
            for (π, γ) ∈ inv-links_i(π, γ') do
              M.collect(π, γ, c, i)
            end
          end
        end
        M = set.new()
        for (π, γ) ∈ R.dom() do
            c = get-upd-com(π, γ, M.get-collection(π, γ, 1), ...,
                            M.get-collection(π, γ, n))
            v = R(π, γ)
            R.set-val(π, γ, apply(v, c))
         in M.add((π, γ, v, c))
        end
      in M.content()
      end
   end
```

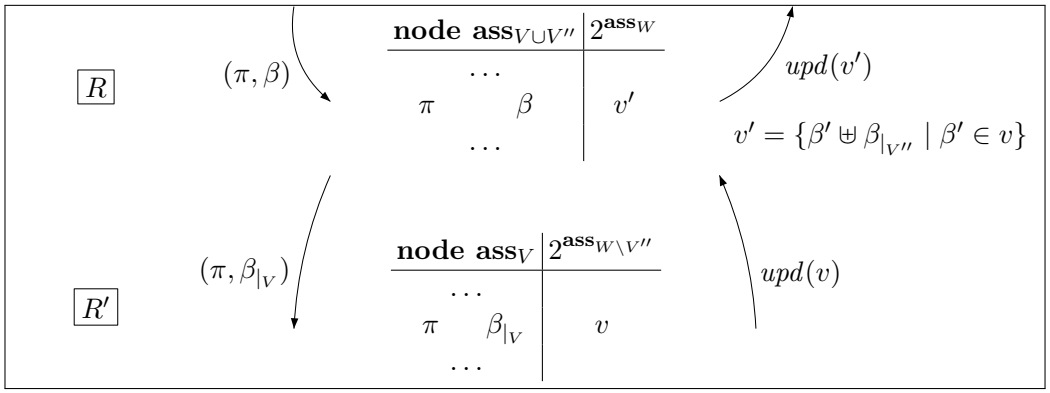
Figure 11.11: Generator of update functions.

```
// Let A'_s be a program evaluator for E ▷ V → V'

A_s = let // program evaluator for E ▷ V ∪ V'' → V'.
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   W = V ∪ V' ∪ V''
   R = registration.new(W, TYPE(E))
   link = fun(π, α) α|_{W\V''} end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c') c' end
   is-done = fun(π, α) is-done'(π, link(π, α)) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.12: A program evaluator for submode rule 1 on streams s.

Whenever the environment of A_s requires to compute the value of E for a new instance, consisting of a possible open start node π and some possible open W-assignment α, then $R.\mathit{add\text{-}inst}(\pi, \alpha)$ is called and adds $R.\mathit{get\text{-}val}(\pi, \alpha) = \bullet$. Then the subregistration is called with $R'.\mathit{add\text{-}inst}(\pi, \alpha_{|_{W \setminus V''}})$ to add a linked entry $R'.\mathit{get\text{-}val}(\pi, \alpha_{|_{W \setminus V''}}) = \bullet$. We then have to ensure that $R.\mathit{get\text{-}val}(\pi, \alpha) = R'.\mathit{get\text{-}val}(\pi, \alpha_{|W \setminus V''})$ how so ever the open value will be instantiated by R' later on: When calling $A_s.\mathit{upd\text{-}val}$ the updates to values in R' by A'_s are also carried out to corresponding values of R. This idea can be implemented by the following program evaluator A_s based on algorithm A'_s in Figure 11.12. There, function *get-upd-com* performs updates c' to values in R, when A'_s performs updates c' to corresponding values in R'.

Let B'_s be a query evaluator that evaluates $E \triangleright V \to V'$ on streams while generating values for V'. We have to define a streaming query evaluator B_s for $E \triangleright V \cup V'' \to V'$ that generates values for V'. The idea is similar to that of A_s but now the registrations updated by B_s and B'_s have the following forms.



For computing the value of $R.\mathit{get\text{-}val}(\pi, \beta)$ by B_s a corresponding entry $R.\mathit{get\text{-}val}(\pi, \beta_{|V})$

```
// Let B'_s query evaluator for E ▷ V → V'

B_s = let // query evaluator for E ▷ V ∪ V'' → V'.
  (get-val', is-done', remove', add-inst', upd-val') = B'_s
  W = V ∪ V' ∪ V''
  R = registration.new(W \ V', TYPE(E))
  link = fun(π, β) β|_V end
  get-val = fun(π, β) R(π, β) end
  letrec
   get-upd-com =
    fun(π, β, c)
      case c
      of none then none
      of c1&c2 then get-upd-com(π, β, c1)&get-upd-com(π, β, c2)
      of extend-set(v) then extend-set({β' ⊎ β|_V'' | β' ∈ v})
      of upd-elem-set(e, c') then upd-elem-set(e ⊎ β|_V'', c')}
      of resolve-elem-set(e, b) then resolve-elem-set(e ⊎ β|_V'', b)}
    end
  in
  is-done = fun(π, β) is-done'(π, link(π, β)) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

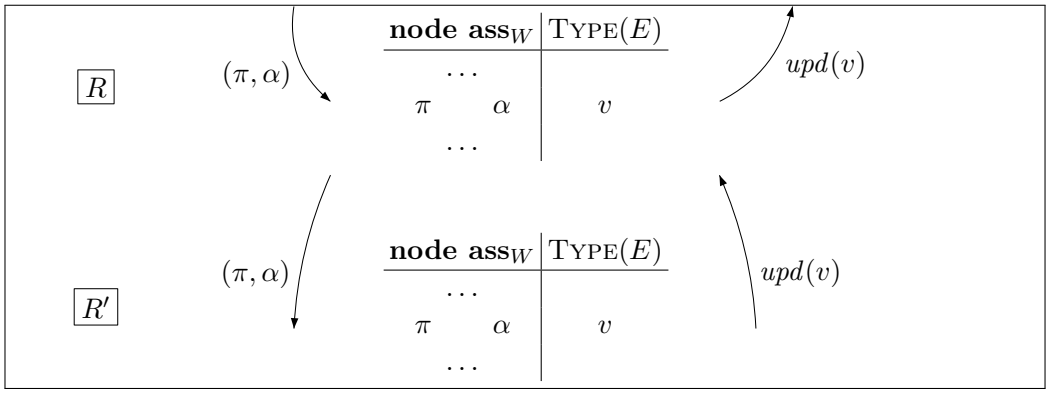
Figure 11.13: A query evaluator for submode rule 1 on streams s.

is added to the registration of B'_s. Whenever some variables assignment β' is added to $R'.get\text{-}val(\pi, \beta_{|V})$, the corresponding variable assignment $\beta' \uplus \beta_{|V''}$ is added to $R.get\text{-}val(\pi, \beta)$. When an existing element e in $R'.get\text{-}val(\pi, \beta_{|V})$ is updated, the corresponding element $e \uplus \beta_{|V''}$ in $R.get\text{-}val(\pi, \beta)$ is updated as well, and similar for resolving elements, as described by function *get-upd-com* of the streaming query evaluator given in Figure 11.13.

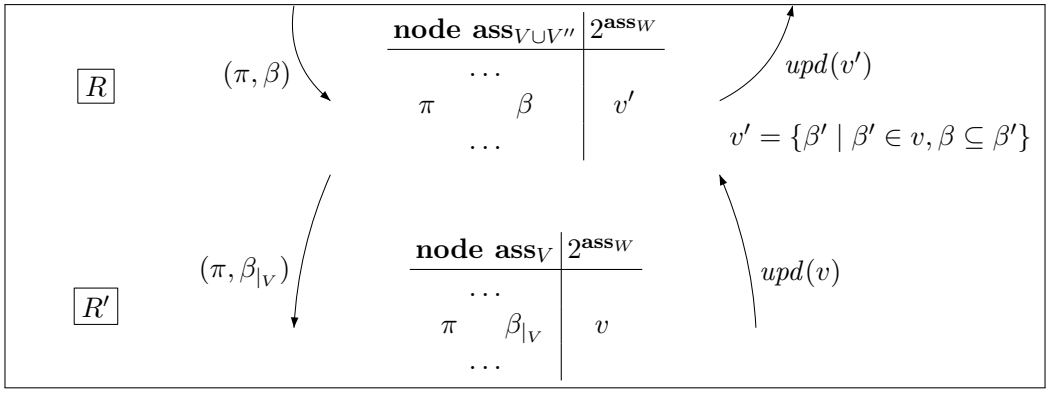
Submode rule 2. Next consider the case, where the judgement was inferred by the second rule for submodes:

$$\frac{E \triangleright V \to V' \cup V''}{E \triangleright V \cup V'' \to V'}$$

Let A'_s be a program evaluator for $E \triangleright V \to V' \cup V''$. We define the program evaluator A_s for $E \triangleright V \cup V'' \to V'$ in Figure 11.14. The registrations R and R' have the following forms where $W = V \cup V' \cup V''$.



Updates to values in R' are also carried out to corresponding values in R, see definition of the *get-upd-com* function. Let B'_s be a query evaluator for $E \triangleright V \to V' \cup V''$. We define the query evaluator B_s for $E \triangleright V \cup V'' \to V'$ in Figure 11.15, with registrations:



Whenever a new variable assignment β' is added to $R'.get\text{-}val(\pi, \beta_{|V})$, then one adds it also $R.get\text{-}val(\pi, \beta)$, but only if $\beta \subseteq \beta'$. Function *get-upd-com* also describes,

```
// Let A'_s be a program evaluator for E ▷ V → V' ∪ V''

A_s = let // program evaluator for E ▷ V ∪ V'' → V'
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   W = V ∪ V' ∪ V''
   R = registration.new(W, TYPE(E))
   link = fun(π, α) α end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c') c' end
   is-done = fun(π, α) is-done'(π, α) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

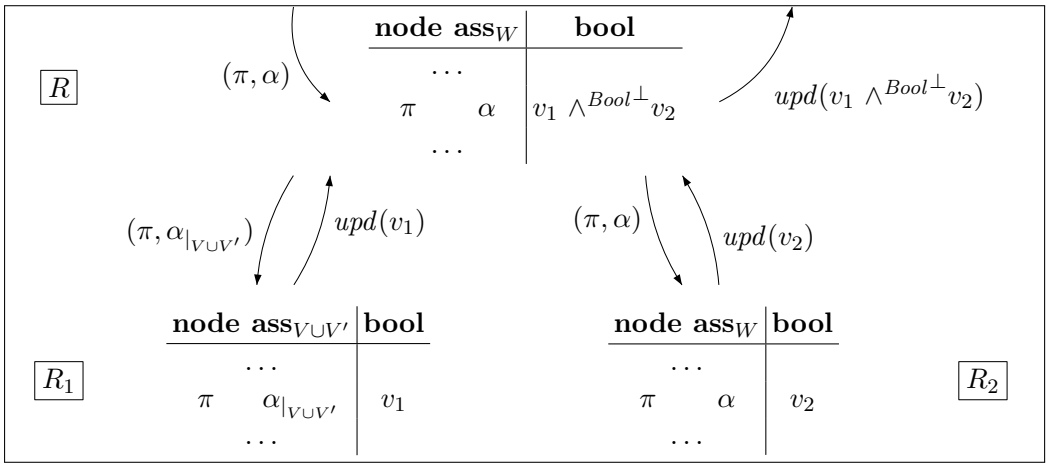
Figure 11.14: A program evaluator for submode rule 2 on streams s.

that updating or resolving the value of an element in $R'.get\text{-}val(\pi, \beta_{|_V})$ results in a update of the respective element in $R.get\text{-}val(\pi, \beta)$ if it exists.

Conjunction. The mode is inferred by inference rule:

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \cup V' \to V'' \quad E = E_1 \wedge E_2}{E \triangleright V \to V' \cup V''}$$

Let A_{1s} be a program evaluator for $E_1 \triangleright V \to V'$ and A_{2s} be a program evaluator for $E_2 \triangleright V \cup V' \to V''$. We define a program evaluator A_s for $E \triangleright V \to V' \cup V''$ in Figure 11.16, while registration have the following form where $W = V \cup V' \cup V''$:



Given some entry $(\pi, \alpha) \in R.dom()$ one waits until entries $(\pi, \alpha_{|_{V \cup V'}}) \in R_1.dom()$ and $(\pi, \alpha) \in R_2.dom()$ have determined values v_1 and v_2, respectively, such that the conjunction $v_1 \wedge^{Bool^\perp} v_2$ can be computed. The *get-upd-com* function implements this by returning *none* until at some event e it holds that *is-done*(π, α) is *true* and

```
// Let B'_s query evaluator for E ▷ V → V' ∪ V''

B_s = let // query evaluator for E ▷ V ∪ V'' → V'
  (get-val', is-done', remove', add-inst', upd-val') = B'_s
  W = V ∪ V' ∪ V''
  R = registration.new(W \ V', TYPE(E))
  link = fun(π, β) β|_V end
  get-val = fun(π, β) R(π, β) end
  letrec
   get-upd-com =
    fun(π, β, c)
      case c
      of none then none
      of c1&c2 then get-upd-com(π, β, c1)&get-upd-com(π, β, c2)
      of extend-set(v) then extend-set({β' | β' ∈ v, β ⊆ β'})
      of upd-elem-set(e, c') then if e ∈ get-val(π, β) then upd-elem-set(e, c') else
          none
      of resolve-elem-set(e, b) then if e ∈ get-val(π, β) then resolve-elem-set(e, b)
          else none
    end
  in
  is-done = fun(π, β) is-done'(π, link(π, β)) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

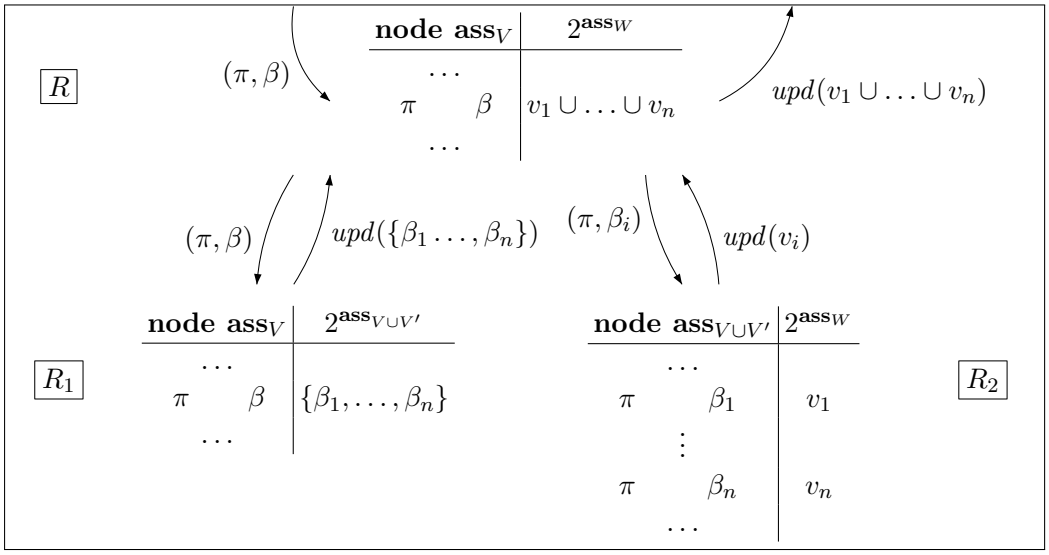
Figure 11.15: A query evaluator for submode rule 2 on streams s.

```
// let A_{1s} a program evaluator for E_1 ▷ V → V'
// let A_{2s} a program evaluator for E_2 ▷ V ∪ V' → V''

A_s = let // program evaluator for E = E_1 ∧ E_2 ▷ V → V' ∪ V''
   (get-val_1, is-done_1, remove_1, add-inst_1, upd-val_1) = A_{1s}
   (get-val_2, is-done_2, remove_2, add-inst_2, upd-val_2) = A_{2s}
   W = V ∪ V' ∪ V''
   R = registration.new(W, TYPE(E))
   link_1 = fun(π, α) α|_{V∪V'} end
   link_2 = fun(π, α) α end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c_1, c_2) if (c_1 ≠ none ∨ c_2 ≠ none) ∧ is-done(π, α) then
        instantiate(get-val_1(π, link_1(π, α)) ∧^{Bool⊥} get-val_2(π, link_2(π, α))) else none end
   is-done = fun(π, α) is-done_1(π, link_1(π, α)) ∧ is-done_2(π, link_2(π, α)) end
   remove = gen-remove(R, remove_1, remove_2, link_1, link_2)
   add-inst = gen-add-inst(R, add-inst_1, add-inst_2, link_1, link_2, get-upd-com)
   upd-val = gen-upd-val(R, upd-val_1, upd-val_2, link_1, link_2, get-upd-com,
                         gen-ass-com(link_1), gen-ass-com(link_2))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.16: A program evaluator for conjunctions on streams s.

such that either v_1 in R_1 was updated by some update command $c_1 \neq none$ or v_2 was updated by some update command $c_2 \neq none$.

Let B_{1s} be a query evaluator for $E_1 \triangleright V \to V'$ and B_{2s} be a query evaluator for $E_2 \triangleright V \cup V' \to V''$. A query evaluator B_s for $E \triangleright V \to V' \cup V''$ is given in Figure 11.17 and Figure 11.18. Its registration have the following form:



The idea is the following. For every entry $(\pi, \beta) \in R.dom()$ there exists a corresponding entry in $(\pi, \beta) \in R_1.dom()$, which computes variable extensions $\{\beta_1, \ldots, \beta_n\}$. According to the mode rule $E_2 \triangleright V \cup V' \to V''$ the evaluator B_{2s}

must receive as input values for variables in V and V'. Therefore, for every β_i where $1 \leq i \leq n$ an entry (π, β_i) is created in R_2, while the union of their answers v_1 until v_n becomes the final answer for B_s. Since the link definition $link_2$ links one entry of R to many entries in R_2, unlike all other evaluators presented here, the query evaluator presented in Figure 11.17 and Figure 11.18 cannot use the common functions that are generated by the function generators in Section 11.2.1. The *add-inst* function of Figure 11.17 therefore starts $add\text{-}inst_1$ for some open node π and open variable assignment β, and calls function *create-entries* that creates for every new answer v_i of R_1 with respect to π and β an entry in R_2, while the result update command is a conjunction $add\text{-}inst_2(\pi, v_1) \& \ldots \& add\text{-}inst_2(\pi, v_n)$. Function *upd-val* presented in Figure 11.18 updates at some event $e \in events(s)$ with variable assignment updates $AssUps$ the entries of R as follows. Variable assignment update in $AssUps$ are carried out to R via function *upd-ass*. Next, before calling function $upd\text{-}val_1$, one starts to compute the new variable assignment updates $AssUps'$ needed for the evaluation of $upd\text{-}val_2$. Hence every existing entries in $AssUps$ are changed according to link definition $link_2$. Next, we obtain updates $U_1 = upd\text{-}val_1(e, AssUps)$, that allows to add to $AssUps'$ those assignment updates for variables in V' that B_{2s} generated and for which R_2 contains corresponding entries. This is done by function *add-ups* which in case that B_{1s} performed an update c' to some variable assignment β' in $upd\text{-}elem\text{-}set(\beta', c')$ calls function *add-var-ass-ups* which in turn adds a respective variable assignment update to $AssUps'$, when c' was an update to some variable in assignment β'. Otherwise, function *add-ups* in the case that B_{1s} removed some element β' in $resolve\text{-}elem\text{-}set(\beta'^{?}, false)$ then the corresponding entry in R_2 is removed, while B_s has to remove the elements of $get\text{-}val_2(\pi, b)$ in $get\text{-}val(\pi)$. Finally, we have compute the variable assignment updates $AssUps'$ and function *upd-val* proceeds to call $upd\text{-}val_2$ with e and $AssUps'$ to obtain U_2. Next we need to add missing updates to M. Similar to function *add-inst* one creates new entries in R_2 for those new variable assignments found by B_{1s} in U_1 by calling function *create-entries* of before. Lastly, we add to M all updates in U_2 by B_{2s} which are also updates for B_s and the function returns M.

Disjunction.

$$\frac{E_1 \triangleright V \to V' \quad E_2 \triangleright V \to V' \quad E = E_1 \vee E_2}{E \triangleright V \to V'}$$

Let A_{is} be program evaluators for $E_i \triangleright V \to V'$ for $i \in \{1, 2\}$. The program evaluator A_s for $E \triangleright V \to V'$ is given in Figure 11.19 and registrations have the following form where $W = V \cup V'$:

```
// Let B_{1s} query evaluator for E_1 ▷ V → V'
// Let B_{2s} query evaluator for E_2 ▷ V ∪ V' → V''

B_s = let // define B_s query evaluator E = E_1 ∧ E_2 ▷ V → V' ∪ V''
  (get-val_1, is-done_1, remove_1, add-inst_1, upd-val_1) = B_{1s}
  (get-val_2, is-done_2, remove_2, add-inst_2, upd-val_2) = B_{2s}
  W = V ∪ V' ∪ V''
  R = registration.new(V, TYPE(E))
  link_1 = fun(π, β) β end
  link_2 = fun(π, β) get-val_1(π, link_1(π, β))
  get-val = fun(π, β) R(π, β) end
  is-done = fun(π, β) is-done_1(π, β) ∧ ∀β'_1 ∈ get-val_1(π, β) : is-done_2(π, β'_1) end
  remove = proc(π, β)
    remove_1
    for β' ∈ link_2(π, β) do
      remove_2(π, β')
    R.remove(π, β)
  end
  letrec
   create-entries =
    fun(π, β, c)
      case c
      of none then none
      of c_1&c_2 then get-upd-com(π, β, c_1)&get-upd-com(π, β, c_2)
      of extend-set(v) then
          {v_1, ..., v_n} = v
       in
          add-inst_2(π, v_1)& ... &add-inst_2(π, v_n)
      else none
    end
  in
  add-inst = fun(π, β)
    R.add-inst(π, β)
    c_1 = add-inst_1(π, β)
    c = create-entries(π, β, c_1)
    R.set-val(π, β, apply(R(π, β), c))
   in
     c
  end
```

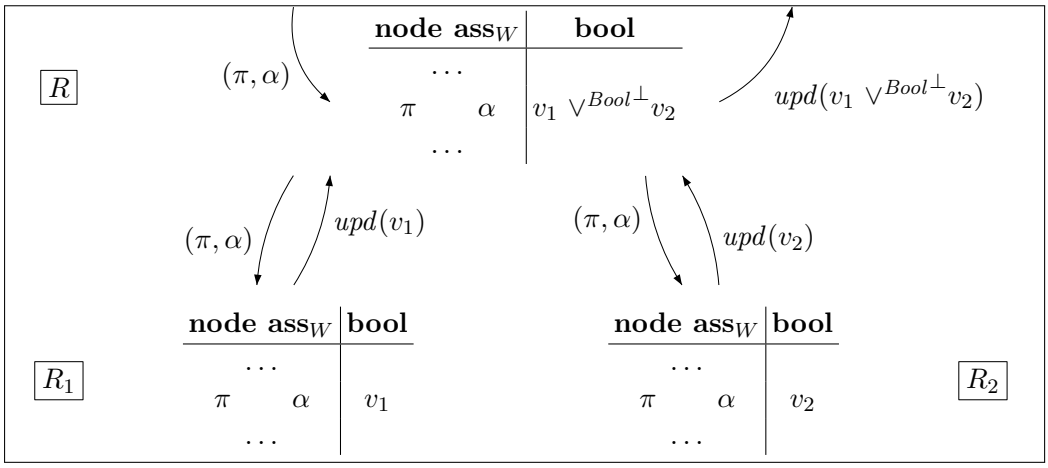
Figure 11.17: Part 1 of a query evaluator for conjunction on streams s.

```
letrec
 add-var-ass-ups = proc(π, β', c', AssUps')
     case c'
     of c1&c2 then
          add-var-ass-ups(π, β', c1, AssUps')&add-var-ass-ups(π, β', c2, AssUps')
     of upd-var-ass(x, v, c) then AssUps'.add(π, β', x, c)
     else return
  end
 add-ups = proc(π, β, c, AssUps', M)
     case c
     of c1&c2 then add-ups(π, β, c1, x)&add-ups(π, β, c2, x)
     of upd-elem-set(β', c') then add-var-ass-ups(π, β', c', AssUps')
     of resolve-elem-set(β'?, false) then
        {β''1, ..., β''n} = get-val2(π, β')
        remove2(π, β')
        M.add(π, β, get-val(π, β),
                 resolve-elem-set(β''1, false)& ... &resolve-elem-set(β''n, false))
     else return
  end
in
upd-val = fun(e ∈ events(s), AssUps)
    for (π, β, x, c) ∈ AssUps do R.upd-ass(π, β, β|W\{x} ⊎ [x/apply(β(x), c)]) end
    AssUps' = set.new()
    for (π, β, x, c) ∈ AssUps do
      for β' ∈ link2(π, β) do AssUps'.add(π, β', x, c) end
    end
    U1 = upd-val1(e, AssUps)
    M = set.new()
    for (π, β, v, c) ∈ U1 do add-ups(π, β, c, AssUps', M) end
    U2 = upd-val2(e, AssUps'.content())
    for (π, β, v, c) ∈ U1 do
      c' = create-entries(π, β, c)
      M.add((π, β, get-val(π, β), c'))
    end
    for (π, β', v, c) ∈ U2 do
      if ∃(π, β'|V, v, c') ∈ M then M.replace((π, β'|V, v, c'), (π, β'|V, v, c'&c))
      else M.add(π, β'|V, v, c))
    end
 in M.content()
end
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.18: Part 2 of a query evaluator for conjunction on streams s.

$$
\begin{array}{l}
\textit{// Let } A_{is} \textit{ be a program evaluator for } E_i \triangleright V \to V',\ 1 \le i \le 2 \\
\\
A_s = \textbf{let}\ \textit{// } A_s \textit{ program evaluator for } E = E_1 \vee E_2 \triangleright V \to V' \\
\quad (\textit{get-val}_i, \textit{is-done}_i, \textit{remove}_i, \textit{add-inst}_i, \textit{upd-val}_i) = A_{is},\ 1 \le i \le 2 \\
\quad W = V \cup V' \\
\quad R = \textit{registration.new}(W, \textsc{Type}(E)) \\
\quad \textit{link}_1 = \textbf{fun}\,(\pi, \alpha)\ \ \alpha\ \ \textbf{end} \\
\quad \textit{link}_2 = \textbf{fun}\,(\pi, \alpha)\ \ \alpha\ \ \textbf{end} \\
\quad \textit{get-val} = \textbf{fun}\,(\pi, \alpha)\ \ R(\pi, \alpha)\ \ \textbf{end} \\
\quad \textit{get-upd-com} = \textbf{fun}\,(\pi, \alpha, c_1, c_2)\ \ \textbf{if}\ \ (c_1 \neq \textit{none} \vee c_2 \neq \textit{none}) \wedge \textit{is-done}(\pi, \alpha)\ \ \textbf{then} \\
\qquad \textit{instantiate}(\textit{get-val}_1(\pi, \alpha) \vee^{Bool^\perp} \textit{get-val}_2(\pi, \alpha))\ \ \textbf{else}\ \ \textit{none}\ \ \textbf{end} \\
\quad \textit{is-done} = \textbf{fun}\,(\pi, \alpha)\ \ \textit{is-done}_1(\pi, \textit{link}_1(\pi, \alpha)) \wedge \textit{is-done}_2(\pi, \alpha)\ \ \textbf{end} \\
\quad \textit{remove} = \textit{gen-remove}(R, \textit{remove}_1, \textit{remove}_2, \textit{link}_1, \textit{link}_2) \\
\quad \textit{add-inst} = \textit{gen-add-inst}(R, \textit{add-inst}_1, \textit{add-inst}_2, \textit{link}_1, \textit{link}_2, \textit{get-upd-com}) \\
\quad \textit{upd-val} = \textit{gen-upd-val}(R, \textit{upd-val}_1, \textit{upd-val}_2, \textit{link}_1, \textit{link}_2, \textit{get-upd-com}, \\
\qquad\qquad \textit{gen-ass-com}(\textit{link}_1), \textit{gen-ass-com}(\textit{link}_2)) \\
\textbf{in}\ \ (\textit{get-val}, \textit{is-done}, \textit{remove}, \textit{add-inst}, \textit{upd-val})
\end{array}
$$

Figure 11.19: A program evaluator for disjunctions on streams s.

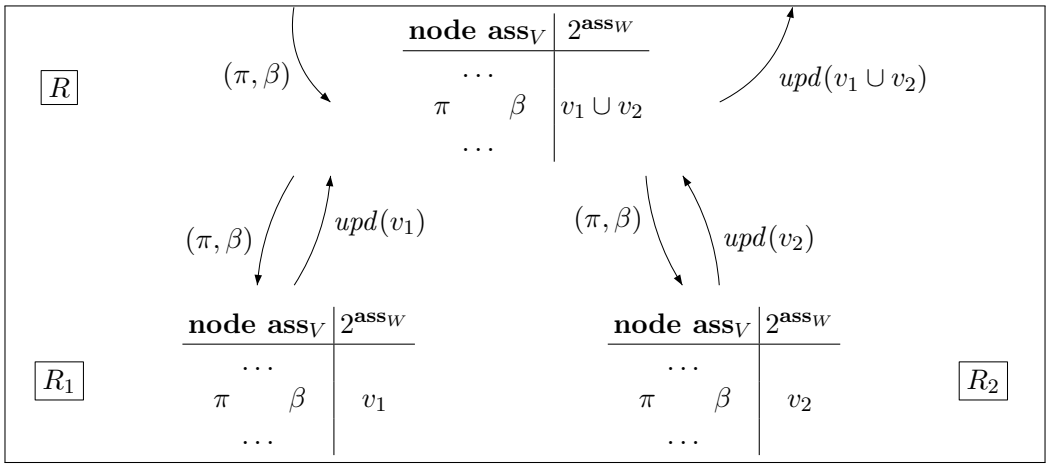


Similar to the program evaluator for conjunction, the program evaluator for the disjunction waits until the event at which both A_{1s} and A_{2s} have finished computing their boolean values, such that function *get-upd-com* can compute the disjunction for the respective entries.

Let B_{is} be query evaluators for $E_i \triangleright V \to V'$ for $i \in \{1, 2\}$. The query evaluator B_s for $E \triangleright V \to V'$ is given in Figure 11.20. Registrations have the following form where $W = V \cup V'$:

```
// Let B_is query evaluator for E_i ▷ V → V', 1 ≤ i ≤ 2

B_s = let // define B_s query evaluator for E = E_1 ∨ E_2 ▷ V → V'
   (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = B_is , 1 ≤ i ≤ 2
   W = V ∪ V'
   R = registration.new(V, TYPE(E))
   link_1 = fun(π, β) β end
   link_2 = fun(π, β) β end
   get-val = fun(π, β) R(π, β) end
   get-upd-com = fun(π, β, c_1, c_2) c_1&c_2 end
   is-done = fun(π, β) is-done_1(π, β) ∧ is-done_2(π, β) end
   remove = gen-remove(R, remove_1, remove_2, link_1, link_2)
   add-inst = gen-add-inst(R, add-inst_1, add-inst_2, link_1, link_2, get-upd-com)
   upd-val = gen-upd-val(R, upd-val_1, upd-val_2, link_1, link_2, get-upd-com,
                         gen-ass-com(link_1), gen-ass-com(link_2))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.20: A query evaluator for disjunctions on streams s.



Given some open node π and variable assignment β, evaluator B_s is in charge of computing the union of variables assignments v_1 and v_2 by evaluators B_{1s} and B_{2s}, respectively, for the linked entries. Therefore, as one sees in function *get-upd-com*, any update to v_1 by c_1 and any update to v_2 by c_2 becomes an update $c_1\&c_2$ to the union $v_1 \cup v_2$.

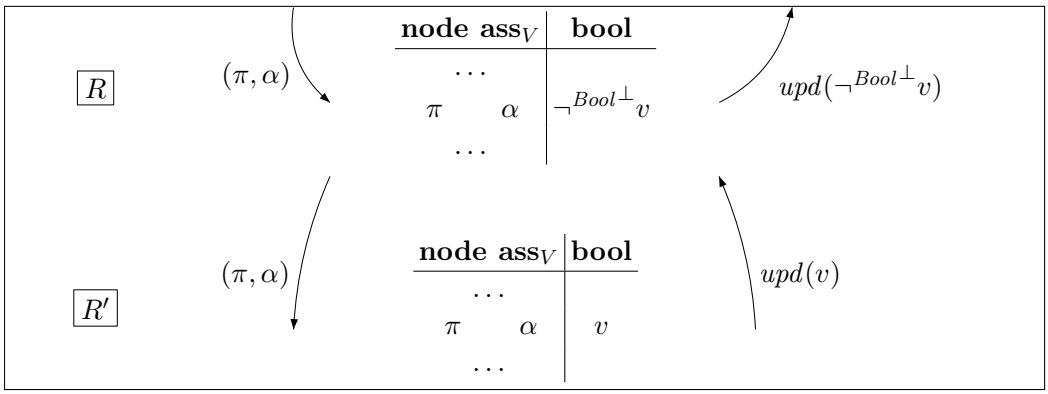
Negation.

$$\frac{E' \triangleright V \to \emptyset \quad E = \neg E'}{E \triangleright V \to \emptyset}$$

Let A'_s be a program evaluator for $E' \triangleright V \to \emptyset$. We define a program evaluator A_s for $E \triangleright V \to \emptyset$ in Figure 11.21. Registrations have the following form:

```
// Let A'_s be a program evaluator for E' ▷ V → ∅

A_s = let // program evaluator for E = ¬E' ▷ V → ∅.
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   R = registration.new(V, TYPE(E))
   link = fun(π, α) α end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c) if c ≠ none ∧ is-done(π, α)) then
        instantiate(¬^Bool⊥ get-val'(π, link(α))) else none end
   is-done = fun(π, α) is-done'(π, α) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.21: A program evaluator for negation on streams s.

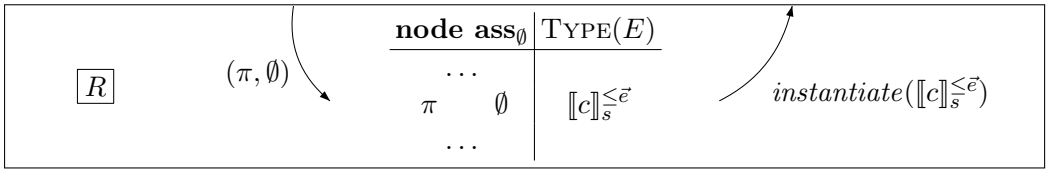


Since the program evaluator computes the negation of the boolean value computed by A'_s, we wait until A'_s has finished computing the boolean value: For some entry $(\pi, \alpha) \in R.dom()$ function *get-upd-com* tests whether the respective update c by A'_s was not empty and whether its computation finished. In this case, one returns the update command $instantiate(\neg^{Bool^\perp} get\text{-}val'(\pi, link(\alpha)))$, which is used in *upd-val* to compute the result value. Since no variables are generated, a query evaluator B_s can be defined from A_s.

Constants.

$$\frac{true}{c \triangleright \emptyset \to \emptyset}$$

We define in Figure 11.22 a program evaluator A_s for $c \triangleright \emptyset \to \emptyset$ where c is an atomic value, the empty set $\emptyset$ or empty list *nil*. Registration R has the following form:



```
// define A_s a program evaluator for E = c ▷ ∅ → ∅.

A_s = let
   R = registration.new(∅, TYPE(E))
   get-val = fun(π, ∅) R(π, ∅) end
   is-done = fun(π, ∅) True end
   remove = gen-remove(R)
   add-inst = fun(π, ∅)
       R.add-inst(π, ∅)
       R.set-val(π, ∅, [[c]]_s^{≤ē})
    in
       instantiate([[c]]_s^{≤ē})
   end
   upd-val = fun(e ∈ events(s), AssUps) ∅ end
in (get-val, is-done, remove, add-inst, upd-val)
```

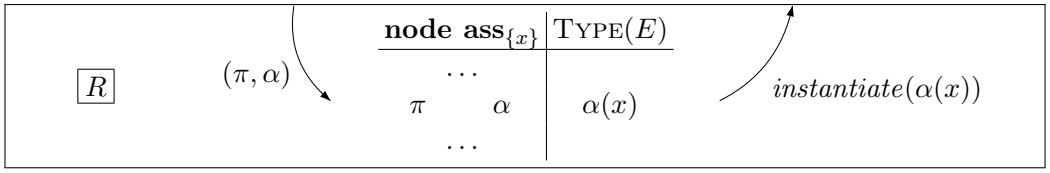
Figure 11.22: A program evaluator for constants on streams s.

The *add-inst* function sets directly the final answer, for which for all entries $\textit{is-done}(\pi, \emptyset) = \textit{true}$. The update function therefore returns always the empty set of updates. Since, no variables are generated, a query evaluator B_s can be defined from A_s. We assume the existence of streaming evaluators for all other constants $c \in \mathcal{C}onst$ of λXP. For example, a program evaluator for function constant *set2list* receives entries (π, α) where $\alpha : \mathbf{ass}_{\{\text{arg}\}}$, which contain the argument set to be transformed into a list, which may possibly contain candidates which are to be decided later. As the argument set is updated, one updates the result list. This requires to store a mapping from elements of the input set to indices in the result list, such that updates to elements of the set can be mapped to updates that change the element at a certain position in the result list.

Variables.

$$\frac{\textit{true}}{x \triangleright \{x\} \to \emptyset}$$

We define a program evaluator A_s for $x \triangleright \{x\} \to \emptyset$ in Figure 11.23, while the registrations has the following form:



Similar to the program evaluator for constants, the *add-inst* functions can directly set the answer $\alpha(x)$ of the query for any instance π and variable assignment α. However, the update function that inputs an event $e \in \textit{events}(s)$ and a set of variable assignment updates *AssUps* may contain a tuple (π, α, x, c) which instructs that the value for x with respect to entry $(\pi, \alpha) \in R.\textit{dom}()$ needs to be updated by c (as the

```
A_s = let // program evaluator for E = x ▷ {x} → ∅.
  R = registration.new({x}, TYPE(E))
  get-val = fun(π, α) R(π, α) end
  is-done = fun(π, α) True end
  remove = gen-remove(R)
  add-inst = fun(π, α)
      R.add-inst(π, α)
      R.set-val(π, α, α(x))
    in
      instantiate(α(x))
  end
  upd-val = fun(e ∈ events(s), AssUps)
      M = set.new()
      for (π, α, y, c) ∈ AssUps do
        if y = x then
          α' = [x/apply(R(π, α), c)]
          R.add-inst(π, α')
          R.set-val(π, α', α'(x))
          M.add((π, α', R(π, α), c))
      end
    in
      M.content()
  end
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.23: A program evaluator for variables on streams s.

program evaluator for tuple decompositions requires, see function *get-ass-com* of Figure 11.35). Since many parent registration entries may be linked with an entry $(\pi, \alpha) \in R.dom()$, while the variable assignment update may only concern some of the parent registration entries, see generator *gen-ass-com*(*link*) in Section 11.2.1, one creates a new entry $(\pi, \alpha') \in R.dom()$, where α' is the variable assignment where x is assigned the by c updated value of $\alpha(x)$. Since, no variables are generated, a query evaluator B_s can be defined from A_s.

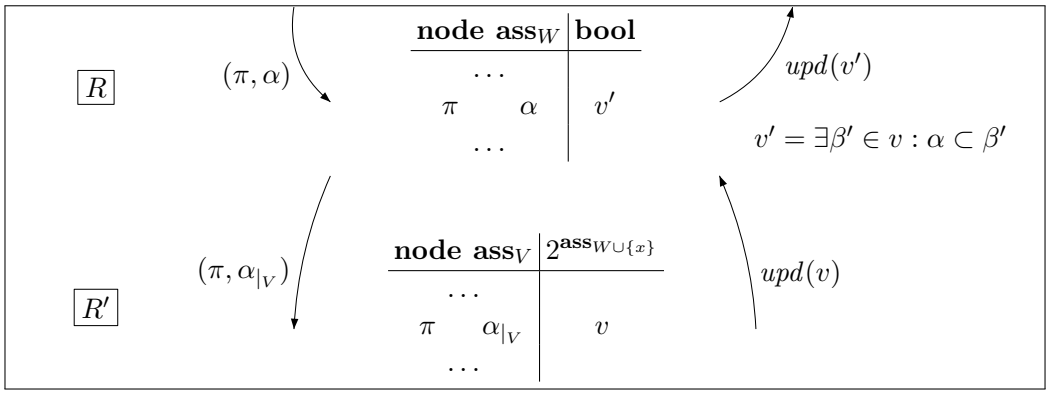
Existential quantification.

$$\frac{E' \triangleright V \to V' \cup \{x\} \quad x \notin V' \quad E = \exists x.E'}{E \triangleright V \to V'}$$

Let B'_s be a query evaluator for $E' \triangleright V \to V' \cup \{x\}$. We define a program evaluator A_s for $E \triangleright V \to V'$ in Figure 11.24. Registrations have the following form where $W = V \cup V'$:

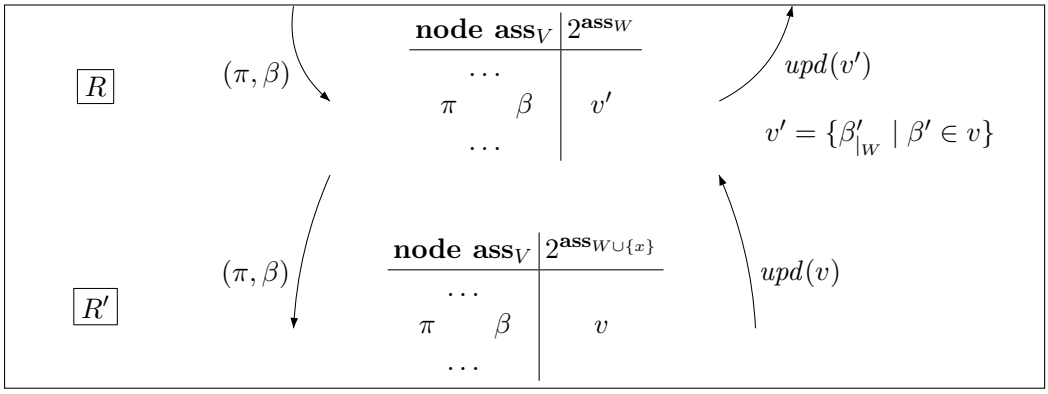
```
// Let B'_s a query evaluator for E' ▷ V → V' ∪ {x}

A_s = let // program evaluator for E = ∃x.E' ▷ V → V'.
  (get-val', is-done', remove', add-inst', upd-val') = B'_s
  W = V ∪ V'
  R = registration.new(W, TYPE(E))
  link = fun(π, α) α_|V end
  get-val = fun(π, α) R(π, α) end
  get-upd-com = fun(π, α, c)
    if c ≠ none ∧ is-done(π, α) then
      if ∃β' ∈ get-val'(π, link(π, α)) : α ⊂ β' then
        instantiate(true)
      else
        instantiate(false)
    else
      none
  end
  is-done = fun(π, α) get-val(π, α) = true ∨ is-done'(π, link(α)) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.24: A program evaluator for existential quantifiers on streams s.



Given some entry $(\pi, \alpha) \in R.dom()$ the evaluator A_s aims to find some variable extensions $\beta' \supset \alpha$ within the set v that is computed by query evaluator B'_s for the linked entry $(\pi, \alpha_{|_V})$ in R'. Function then *get-upd-com* waits until the set v by B'_s has been determined, such that the existence of some $\beta' \in v$ with $\alpha \subset \beta'$ implies that there exists a value for x, which is $\beta'(x)$, for which the formula becomes *true*.

Let B'_s be a query evaluator for $E' \triangleright V \to V' \cup \{x\}$. We define a query evaluator B_s for $E \triangleright V \to V'$ in Figure 11.25. Registrations have the following form where $W = V \cup V'$:



The idea is that query evaluator B'_s computes a set of variable assignment extension v for some β and some node π, which also assigns values for variable x. Query evaluator B_s then computes the set $v' = \{\beta'_{|_W} \mid \beta' \in v\}$ that restricts assignments β' to variables in W without x. The challenge is, given an update command c to variable extensions v from B'_s, how to compute update commands c' by B_s that update v' accordingly. For this, we refer to function *get-upd-com*, which tests the various possible case for update command c. When extending the set with new elements β', then one extends simply the set v' by elements $\beta'_{|_W}$. When resolving an element in v, then one resolves the corresponding element in v'. However, when updating some element $e \in v$ by $\bar{c}$, then $\bar{c}$ may contain updates for values for x in e, which do not concern corresponding values in v'. In this case, we rely on some function *get-upd-var-ass-elem*, which returns the proper adapted update commands, regardless of updates $\bar{c}$ that do not concern elements of v'.

Navigational formulas.

Let E be a simple navigational query with mode $E \triangleright V \to V'$. For navigational queries we will apply our streaming algorithm of Chapter 8 that runs eNwAs. They require however to be evaluated from the root of any tree, for which we consider the slightly adapted formula

$$start(E, z) =_{def} ch^*(is_z \wedge E).$$

When evaluating E at some node $\pi \in \text{nodes}(s)$ with variable assignment α then this is equivalent to evaluate $start(E, z)$ on the root $root(\pi)$ to which π belongs while setting variable z to π:

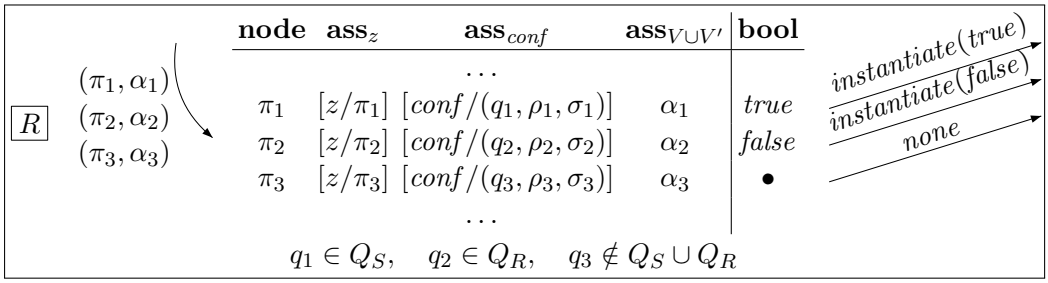
$$[\![E]\!]_{s,\pi,\alpha} = [\![start(E, z)]\!]_{s,root(\pi),\alpha \uplus [z/\pi]}$$

In streaming we next eliminate backward axes in $start(F, z)$ as described in Section 10.1.3. Finally we translate the backward axes free and simple query into an early nested word automata $\mathcal{E} = (\Sigma, \Delta, Q, Q_I, Q_F, \Gamma, \delta, Q_S, Q_R)$. We present a program evaluator A_t for some tree t within the sequence of trees s in Figure 11.26 and Figure 11.27 that runs $\mathcal{E}$ for every event of the stream that belong to t and whose registration stores besides variable assignments for variables in $V \cup V'$ also values for start variable z and configurations of the automata for variable *conf*. Since a

```
// Let B'_s query evaluator for E' ▷ V → V' ∪ {x}
// define B_s query evaluator for E = ∃x.E' ▷ V → V'.

B_s = let
  (get-val', is-done', remove', add-inst', upd-val') = B'_s
  W = V ∪ V'
  R = registration.new(W \ V', TYPE(E))
  link = fun(π, β) β end
  get-val = fun(π, β) R(π, β) end
  letrec
   get-upd-var-ass-elem =
    fun(e, c)
      case c
      of none then none
      of c1&c2 then get-upd-var-ass-elem(e, c1)&get-upd-var-ass-elem(e, c2)
      of extend-var-ass(y, v) then if [y/v] ⊆ e|_W then
           upd-elem-set(e|_W, extend-var-ass(y, v)) else none
      of upd-var-ass(y, v, c̄) then if [y/v] ⊆ e|_W then
           upd-elem-set(e|_W, upd-var-ass(y, v, c̄)) else none
    end
   get-upd-com =
    fun(π, β, c)
      case c
      of none then none
      of c1&c2 then get-upd-com(π, β, c1)&get-upd-com(π, β, c2)
      of extend-set(v) then extend-set({β'|_W | β' ∈ v})
      of upd-elem-set(e, c̄) then get-upd-var-ass-elem(e, c̄)
      of resolve-elem-set(e, b) then resolve-elem-set(e|_W)
    end
  in
  is-done = fun(π, β) is-done'(π, link(β)) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

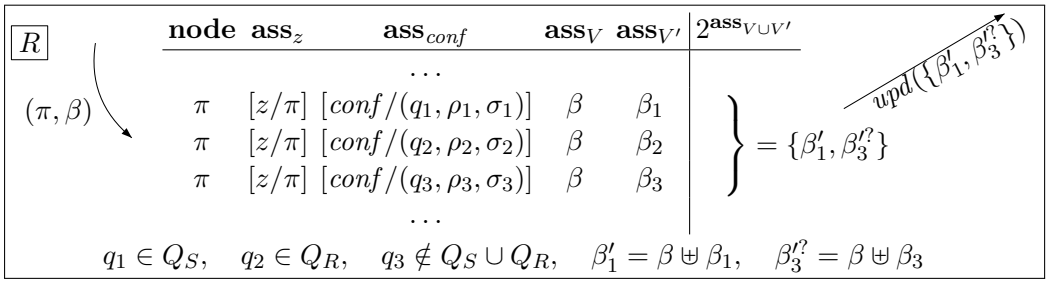
Figure 11.25: A query evaluator for existential quantifiers on streams s.

ENWA runs on one tree only, we maintain program and query evaluators for every tree t in s. However, not all navigational formulas must be run on all trees of the input sequence s. We leave it to static analysis to reduce the number of program and query evaluators for navigational formulas, and otherwise run one evaluator per tree t in s. We refer to Chapter 8 for a more detailed explanation of running automata over streams, while in the following we only explain the essentials. The registration has the following form:



Function *add-inst* that is run for some open node π and variable assignment $\alpha \in \mathbf{ass}_{V \cup V'}$ is called at the opening event of π. The automaton $\mathcal{E}$ guides the creation of entry in R, by testing all variable annotations v that do not lead to some rejection state for the opening event at π. If $z \in v$ then an entry in R is created by calling $R.$*add-inst*, which sets the new configuration and which sets start variable z to π. When no entry was created in R for π and α, then the navigational formula evaluates to *false*, otherwise one analyses the state q for the entry that contains π and α and returns *true* if $q \in Q_S$ is a selection state, one returns *false* if $q \in Q_R$ is a rejection states, and otherwise the $\bullet$- value is stored in R as intermediate answer. In the later case, it is the update function in Figure 11.27 that runs the automaton for these entries at later events. It uses function *test-new-state* that computes updates for those entries whose configuration's state reached a selection or a rejection state.

We present a query evaluator B_t for $E \triangleright V \to V'$ in Figure 11.28 and Figure 11.29 that runs $\mathcal{E}$ that we obtain as explained for the program evaluator. The registration has the following form:



One can see that the registration for the query evaluator contains several entries for one instance $(\pi, \beta) \in R.dom()$. The reason is that the automaton generates all variable assignments for variables in V'. The result for such an instance is

```
// define A_t program evaluator for F ▷ V → V'.
// where conf is a global variable for configurations of ENWA 𝓔

A_t = let
  𝓔 = (Σ, Δ, Q, Q_I, Q_F, Γ, δ, Q_S, Q_R)
  W = {z, conf} ∪ V ∪ V'
  R = registration.new(W, TYPE(E))
  get-val = fun(π, α) R(π, α) end
  is-done = fun(π, α : ass_{V∪V'})
      ∀π, q, ρ, σ, α'. ([conf/(q, ρ, σ), z/π] ⊎ α ∈ R.dom()) → q ∈ Q_S ∪ Q_R
  end
  remove = gen-remove(R)
  add-inst = fun(π, α : ass_{V∪V'}) //call at opening event of π
      a = lab(π)
      for (•, α') ∈ R.dom() : ∀x ∈ α.dom() : α'(x) = • ∨ α'(x) = α(x) do
        (q, ρ, σ) = α'(conf)
        for q --⟨(a,v)⟩:γ--> q' ∈ δ : v ≠ ∅, q' ∉ Q_R do
          if z ∈ v then
            R.add-inst(π, [conf/(q', ρv, σγ) ⊎ α'|_{(V∪V')\v} {[x/π] | x ∈ v})
        end
      end
      M = {(π, α') ∈ R.dom() | α ⊆ α'}
      b = if M = ∅ then false
          else //#M = 1
            (π, α') ∈ M
            (q, ρ, σ) = α'(conf)
            if q ∈ Q_S then true else if q ∈ Q_R then false else •
      c = if b = • then none else instantiate(b)
      R.set-val(π, α, b)
    in c
  end
```

Figure 11.26: Part 1 of a program evaluator for navigational formulas on streams s.

```
test-new-state = fun(q, α, M)
    if q ∈ Q_S ∪ Q_R then
      if q ∈ Q_R then R.remove(α(z), α)
      c = if q ∈ Q_S then instantiate(true) else instantiate(false)
      π = α(z)
      α' = α|_{V∪V'}
      v = R(π, α')
      R.set-val(π, α', apply(v, c))
      M.add((π, α', v, c))
  in M
end
upd-val =
    fun(e ∈ events(s), AssUps)
      (π, c) = e
      if π ∉ nodes(t) then
        for (π, α, x, c) ∈ AssUps do
          R.upd-ass(π, α, α|_{W\{x}} ⊎ [x/apply(α(x), c)]) end
        a = lab(π)
        for (π', α) ∈ R.dom() do
          if c = op then
            (q, ρ, σ) = α(conf)
            for q --⟨(a,v)⟩:γ--> q' ∈ δ : v ≠ ∅, q' ∉ Q_R do
              if z ∉ v then
                alpha' = [conf/(q', ρv, σγ) ⊎ α|_{(V∪V')\v} ⊎ {[x/π] | x ∈ v}]
                R.add-inst(π', α')
                test-new-state(q', α', M)
            end
            if q ∉ Q_S then
              q --⟨(a,∅)⟩:γ--> q' ∈ δ
              α' = R.upd-ass(π', α, α|_{V∪V'∪{z}} ⊎ [conf/(q', ρ∅, σγ)])
              test-new-state(q', α', M)
          else
            if q ∉ Q_S then
              (q, ρv, σγ) = α(conf)
              q --⟨/(a,v)⟩:γ--> q' ∈ δ
              α' = R.upd-ass(π', α, α|_{V∪V'∪{z}} ⊎ [conf/(q', ρ, σ)])
              test-new-state(q', α', M)
          end
        in
          M.content()
        else ∅
      end
in (get-val, is-done, remove, add-inst, upd-val)
```

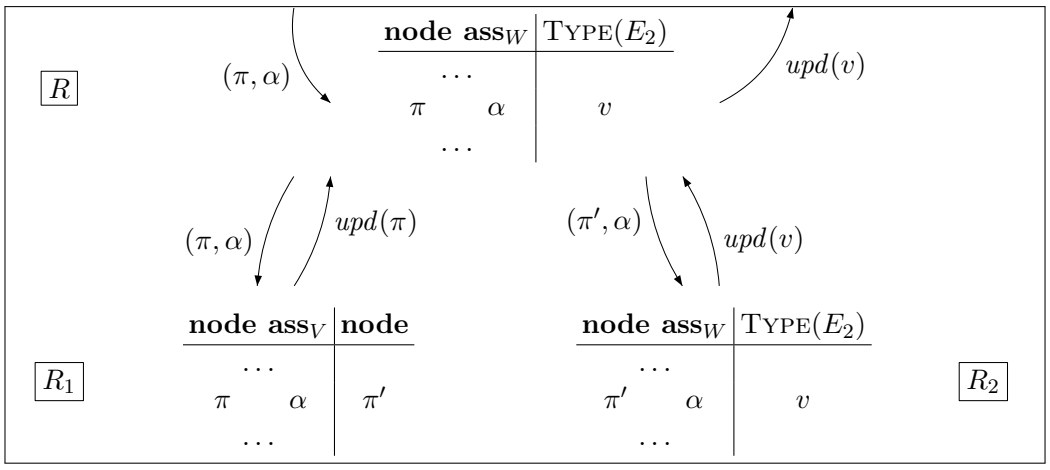
Figure 11.27: Part 2 of a program evaluator for navigational formulas on streams s.

then the set of extended variable assignments that is accumulated from all entries that carry π and β. Similar to the program evaluator, function *add-inst* creates entries for some open node π and variable assignment β under the guidance of the automaton. The result set then contains all non-rejecting variable extensions, where variable extensions with a non-selecting and non-rejecting configuration are candidates whose selection is to be determined later on. Function *upd-val* runs the automaton for all entries, possible creating new entries or candidates belonging to some previously created instance. Function *test-new-state* removes and resolves answer elements for previous entries whose state became selecting or rejecting at the respective event. Function *test-new-candidate* adds new answer elements to the final set, which are possibly candidates whose selection must yet be determined. Finally function *add-com-and-set-val* keeps track that R contains the updated values for created instances, while it adds update commands to answer set M.

Location.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to V' \quad E = \mathbf{at}(E_1, E_2)}{E \triangleright V \to V'}$$

The semantics of formula $\mathbf{at}(E_1, E_2)$ at some open π and variable assignment α in $V \cup V'$, is equal to the semantics of E_2 with the same variable assignment α, but at the node selected by E_1 for π and α. Let A_{1s} be a program evaluator for $E_1 \triangleright V \to \emptyset$ and A_{2s} be a program evaluator for $E_2 \triangleright V \to V'$. The program evaluator A_s for $E \triangleright V \to V'$ is given in Figure 11.30 and Figure 11.31. The registrations have the following form:



Given an instance $(\pi, \alpha) \in R.dom()$ one observes that an answer node π' of the linked instance in R_1 is used as new start node to create an instance $(\pi', \alpha) \in R_2.dom()$. Since the node of linked entries changes, unlike to all other registrations, the program evaluator for location cannot make use of the function generators *gen-ass-com*, *gen-add-inst*, *gen-remove*, and *gen-upd-val* presented in Section 11.2.1, but it mimics the functions that are being generated while taking care of proper linkage with respect to the semantics. A query evaluator for E works just as the program evaluator, but it assumes a query evaluator B_s for $E_2 \triangleright V \to V'$.

```
// define B_t query evaluator for F ▷ V → V'.
// where conf is a global variable for configurations of ENWA ℰ

B_t = let
  ℰ = (Σ, Δ, Q, Q_I, Q_F, Γ, δ, Q_S, Q_R)
  W = {z, conf} ∪ V ∪ V'
  R = registration.new(W, TYPE(E))
  get-val = fun(π, β)  R(π, β)  end
  is-done = fun(π, β : ass_V)
      ∀π, q, ρ, σ, β'. ([conf/(q, ρ, σ), z/π] ⊎ β' ∈ R.dom() ∧ β ⊆ β')  →  q ∈ Q_S ∪ Q_R
  end
  remove = gen-remove(R)
  add-inst =
      fun(π, β : ass_V)  //call at the opening event of π
        a = lab(π)
        for  (•, β') ∈ R.dom() : β'(z) = • ∧ ∀x ∈ β.dom() : β'(x) = • ∨ β'(x) = β(x)  do
          (q, ρ, σ) = β'(conf)
          for  q --⟨(a,v)⟩:γ--> q' ∈ δ :  v ≠ ∅ ,  q' ∉ Q_R  do
            if  z ∈ v  then
              R.add-inst(π, [conf/(q', ρv, σγ)] ⊎ β'|_{(V∪V')\v} ⊎ {[x/π] | x ∈ v})
          end
        end
        M = set.new()
        for  (π, β') ∈ R.dom() : β ⊆ β'  do
          (q, ρ, σ) = β'(conf)
          case  q
          of  q ∈ Q_S  then  M.add(β'|_{V∪V'})
          of  q ∉ Q_S ∪ Q_R  then  M.add(β'?|_{V∪V'})
          of  q ∈ Q_R  then  ⊥
        end
        c = extend-set(M.content())
        R.set-val(π, β, apply(R(π, β), c))
       in  c
      end
  add-com-and-set-val = proc(π, β, M, c)
      v = R(π, β)
      R.set-val(π, β, apply(v, c))
      if  ∃(π, β, v, c') ∈ M  then  M.replace((π, β, v, c'), (π, β, v, c'&c))
      else  M.add((π, β, v, c))
  end
```

Figure 11.28: Part 1 of a query evaluator for navigational formulas on streams s.

```
test-new-state = fun(q, β, M)
    if q ∈ Q_S ∪ Q_R then
      if q ∈ Q_R then R.remove(β(z), β)
      c = if q ∈ Q_S then resolve-elem-set(β?_{|V∪V'}, true)
          else resolve-elem-set(β?_{|V∪V'}, false)
      add-com-and-set-val(β(z), β_{|V}, M, c)
 in M
end
test-new-candidate = fun(q, β, M)  //q ∉ Q_R
    c = if q ∈ Q_S then extend-set({β_{|V∪V'}})
        else extend-set({β?_{|V∪V'}})
    add-com-and-set-val(β(z), β_{|V}, M, c)
 in M
end
upd-val = fun(e ∈ events(s), AssUps)
  (π, c) = e
  if π ∉ nodes(t) then
    for (π, β, x, c) ∈ AssUps do R.upd-ass(π, β, β_{|W\{x}} ⊎ [x/apply(β(x), c)]) end
    a = lab(π)
    M = set.new()
    for (π', β) ∈ R.dom() do
      if c = op then
        (q, ρ, σ) = β(conf)
        for q --⟨(a,v)⟩:γ--> q' ∈ δ: v ≠ ∅, q' ∉ Q_R do
          if z ∉ v then
            β' = [conf/(q', ρv, σγ)] ⊎ β_{|(V∪V')\v} ⊎ {[x/π] | x ∈ v})
            R.add-inst(π', β')
            test-new-candidate(q', β', M)
        end
        if q ∉ Q_S then
          q --⟨(a,∅)⟩:γ--> q' ∈ δ
          β' = R.upd-ass(π', β, β_{|V∪V'∪{z}} ⊎ [conf/(q', ρ∅, σγ)])
          test-new-state(q', β', M)
      else
        if q ∉ Q_S then
          (q, ρv, σγ) = β(conf)
          q --⟨/(a,v)⟩:γ--> q' ∈ δ
          β' = R.upd-ass(π', β, β_{|V∪V'∪{z}} ⊎ [conf/(q', ρ, σ)])
          test-new-state(q', β', M)
      end
     in M.content()
    else ∅
  end
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.29: Part 2 of a query evaluator for navigational formulas on streams s.

```
// let A_{1s} a program evaluator for E_1 ▷ V → ∅
// let A_{2s} a program evaluator for E_2 ▷ V → V'

A_s = let // program evaluator for E = at(E_1, E_2) ▷ V → V'
  (get-val_1, is-done_1, remove_1, add-inst_1, upd-val_1) = A_{1s}
  (get-val_2, is-done_2, remove_2, add-inst_2, upd-val_2) = A_{2s}
  W = V ∪ V'
  R = registration.new(W, TYPE(E))
  link_1 = fun(π, α) α_{|V} end
  link_2 = fun(π, α) α end
  get-val = fun(π, α) R(π, α) end
  is-done = fun(π, α)
      is-done_1(π, link_1(π, α)) ∧ is-done_2(get-val_1(π, link_1(π, α)), link_2(π, α)) end
  remove = proc(π, α)
    remove_1(π, link_1(π, α))
    remove_2(get-val_1(π, link_1(π, α)), link_2(π, α))
    R.remove(π, α)
  end
  add-inst = fun(π, α)
     R.add-inst(π, α)
     c_1 = add-inst_1(π, link_1(π, α))
     c_2 = add-inst_2(get-val_1(π, link_1(π, α)), link_2(π, α))
     R.set-val(π, α, apply(R(π, α), c_2))
   in
     c_2
  end
  get-ass-com_2 = fun(AssUps)
    AssUps' = set.new()
    for (π, α, x, c) ∈ AssUps do
      AssUps'.add(get-val_1(π, link_1(π, α)), link_2(π, α), x, c)
    end
   in
     AssUps'.content()
  end
```

Figure 11.30: Part 1 of a program evaluator for locations on streams s.

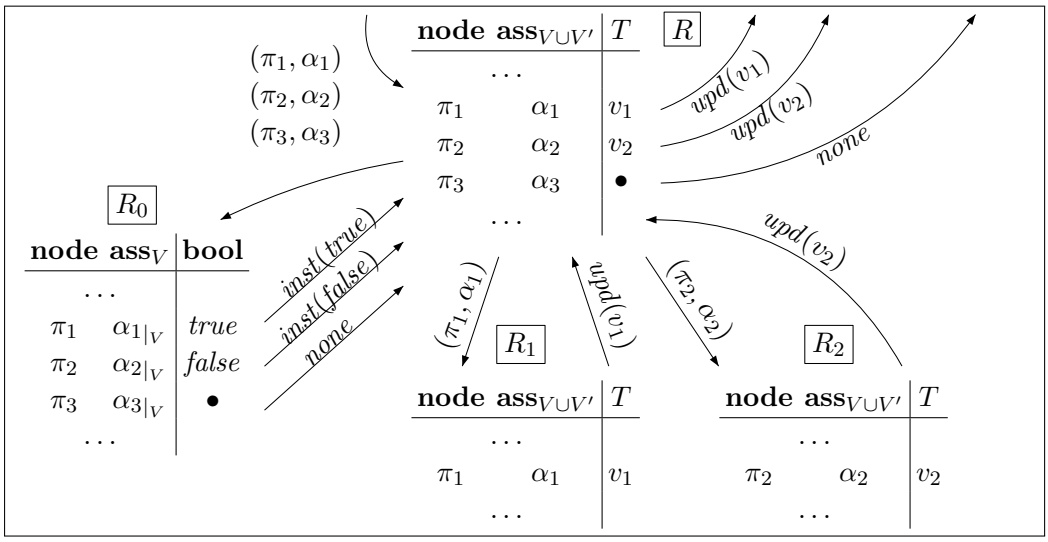
```
letrec
 get-upd-com =
  fun(π, α, c1, c2)
    case c1
    of none then c2
    of instantiate(v) then add-inst2(v, link2(π, α))
  end
in
upd-val = fun(e ∈ events(s), AssUps)
  for (π, α, x, c) ∈ AssUps do R.upd-ass(π, α, α|W\{x} ⊎ [x/apply(α(x), c)])
  U2 = upd-val2(e, get-ass-com2(AssUps))
  M = master-registration.new(R, R1, R2)
  for (π, α) ∈ R.dom() do
    M.init-collection(π, α, 1)
    M.init-collection(π, α, 2)
  end
  for (π′, α′, v, c) ∈ U2 do
    for (π, α) ∈ {(π, α) | get-val1(π, link1(π, α)) = π′, link2(π, α) = α′} do
      M.collect(π, α, c, 2)
    end
  end
  get-ass-com1 = gen-ass-com(link1)
  U1 = upd-val1(e, get-ass-com1(AssUps, ∅))
  for (π, α′, v, c) ∈ U1 do
    for (π, α) ∈ {(π, α) | link1(π, α) = α′} do
      M.collect(π, α, c, 1)
    end
  end
  M = set.new()
  for (π, α) ∈ R.dom() do
      c = get-upd-com(π, α, M.get-collection(π, α, 1), M.get-collection(π, α, 2))
      v = R(π, α)
      R.set-val(π, α, apply(v, c))
    in
      M.add((π, α, v, c))
  end
in
  M.content()
end
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.31: Part 2 of a program evaluator for locations on streams s.

Conditionals.

$$\frac{E_0 \triangleright V \to \emptyset \quad E_1 \triangleright V \to V' \quad E_2 \triangleright V \to V' \quad E = \textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2}{E \triangleright V \to V'}$$

Let A_{0s} be a program evaluator for $E_0 : \textbf{bool} \triangleright V \to \emptyset$ and A_{is} be program evaluators for $E_i : T \triangleright V \to V'$ for $i \in \{1, 2\}$. We define a program evaluator A_s for $E : T \triangleright V \to V'$ in Figure 11.32. Registrations have the following form:



For algorithm A_s in the beginning it is not known whether A_{0s} evaluates to *true*, for which the result by A_{1s} is the answer, or whether A_{0s} evaluates to $False$, for which the result by A_{2s} is needed. Therefore we wait until A_{0s} has determined its answer, at which event we start A_1 or A_2, respectively. Therefore in the example we consider three entries $(\pi_1, \alpha_1), (\pi_2, \alpha_2), (\pi_3, \alpha_3) \in R.dom()$. For entries where A_{0s} returns *true*, as with entry $(\pi_1, \alpha_1) \in R.dom()$, registration R_1 is started, while when A_{0s} returns *false*, then R_2 is started. Due to our restrictions on conditionals from Section 11.1.1 the possible late launch of either A_1 or A_2 does not require information from the past of the stream. The *get-upd-com* function of the program evaluator tests the update c_0 of A_{0s} at some event. If it is *none*, then maybe because it has already determined its answer $\textit{get-val}_0(\pi, \textit{link}_0(\pi, \alpha))$ for some entry $(\pi, \alpha) \in R.dom()$. If it was *true*, then the value of R for the entry requires the same update c_1 with which the corresponding value in R_1 was updated with. If the answer was *false* then one takes update c_2 from R_2, or the answer is not yet known (•), such that neither R_1 nor R_2 has been started for the entry, for which update command *none* is returned. If update c_0 was *instantiate*(v), meaning that R_0 determined its answer, one starts either R_1 or R_2 depending on v. This is a possible late start, e.g. a start at some event $e = (op, \pi')$ that was input to the *upd-val* function, of a node π' that came after π. The query evaluator B_s works similarly and only differs in link definitions.

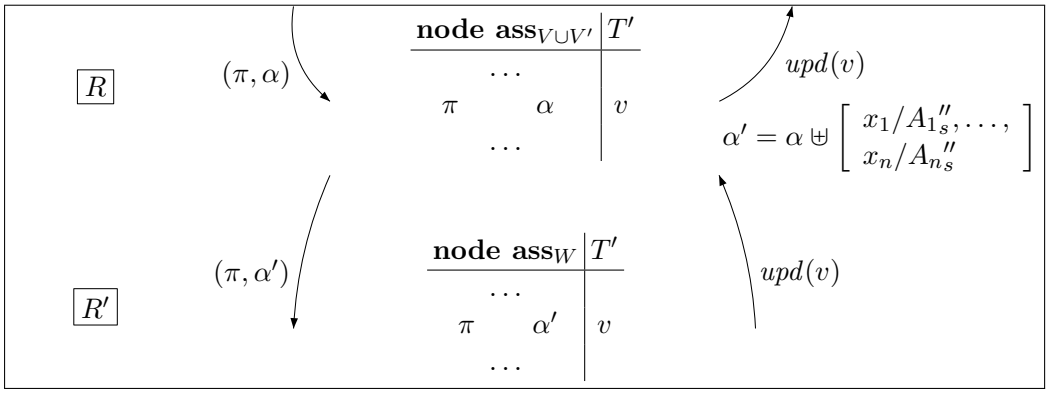
```
// Let A_{1s} be a program evaluator for E_0 ▷ V → ∅
// Let A_{is} be a program evaluator for E_i ▷ V → V' for 1 ≤ i ≤ 2

A_s = let // program evaluator for E = if E_0 then E_1 else E_2 ▷ V → V'
  (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_{is}, for all 0 ≤ i ≤ 2
  R = registration.new(V ∪ V', TYPE(E))
  link_0 = fun(π, α) α_{|V} end
  link_1 = fun(π, α) α end
  link_2 = fun(π, α) α end
  get-val = fun(π, α) R(π, α) end
  get-upd-com = fun(π, α, c_0, c_1, c_2)
      case c_0
      of none then
        case get-val_0(π, link_0(π, α))
        of true then c_1
        of false then c_2
        of • then none
        end
      of instantiate(v) then //starts A_{1s} or A_{2s} possibly late
        if v then add-inst_1(π, link_1(π, α))
        else add-inst_2(π, link_2(π, α))
      end
  end
  is-done = fun(π, α) (get-val_0(π, α_{|V}) = true ∧ is-done_1(π, α))
    ∨(get-val_0(π, α_{|V}) = false ∧ is-done_2(π, α)) end
  remove = gen-remove(R, remove_0, remove_1, remove_2, link_0, link_1, link_2)
  add-inst = fun(π, α)
      R.add-inst(π, α)
      c_0 = add-inst_0(π, link_0(π, α))
      c = get-upd-com(π, α, c_0, none, none)
    in
      c
  end
  upd-val = gen-upd-val(R, upd-val_0, upd-val_1, upd-val_2, link_0, link_1, link_2, get-upd-com,
              gen-ass-com(link_0), gen-ass-com(link_1), gen-ass-com(link_2))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.32: A program evaluator for conditionals on streams s.

Recursions.

$$\frac{\begin{array}{c}\forall 1 \leq i \leq n.\ E_i \triangleright V \cup \{x_1, \ldots, x_n\} \to \emptyset \\ E' \triangleright V \cup \{x_1, \ldots, x_n\} \to V' \\ E = \textbf{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \textbf{in}\ E'\end{array}}{E \triangleright V \to V'}$$

Let A_{is} be a program evaluator for $E_i \triangleright V \cup \{x_1, \ldots, x_n \to \emptyset$, $1 \leq i \leq n.$ and A'_s be a program evaluator for $E' \triangleright V \cup \{x_1, \ldots, x_n\} \to V'$. The program evaluator A_s for $E \triangleright V \to V'$ is given in Figure 11.33. The registrations have the following form where $W = V \cup V' \cup \{x_1, \ldots, x_n\}$:



For some instance $(\pi, \alpha) \in R.dom()$ the respective entry in R' is $\alpha' = \alpha \uplus [x_1/A_{1s}'', \ldots, x_n/A_{ns}'']$ where the A_{is}'' are recursively defined by program evaluators A_{is}. The subevaluators for variables x_i in A'_s that are assigned to A_{is}'' are then the evaluators A_{is}'' themselves. The query evaluator B_s is similar to the program evaluator. It differs only in link definitions.

Tuple constructions.

$$\frac{\forall 1 \leq i \leq n.\ E_i \triangleright V \to \emptyset \quad E = (E_1, \ldots, E_n)}{E \triangleright V \to \emptyset}$$

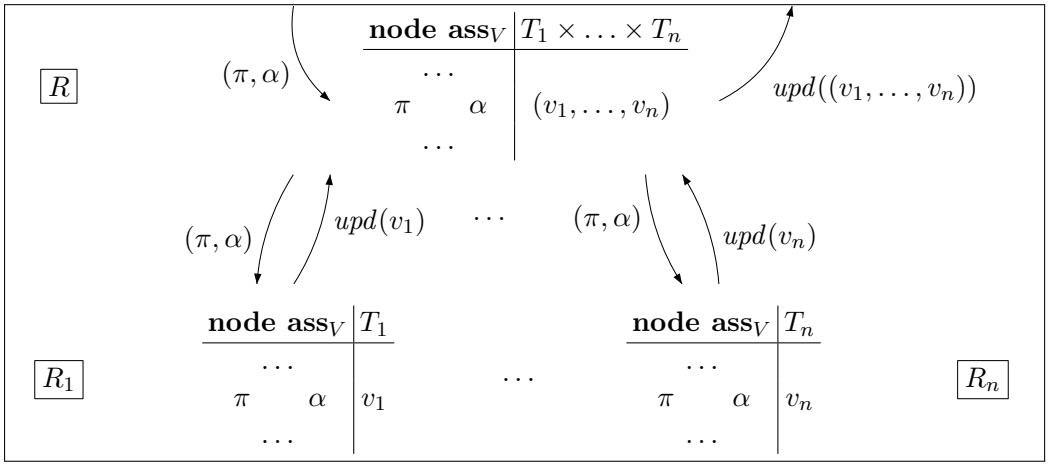
Let A_{is} be program evaluator for $E_i : T_i \triangleright V \to \emptyset$ for $1 \leq i \leq n$. We define a program evaluator A_s for $E \triangleright V \to \emptyset$ in Figure 11.34. Registrations have the following form:

```
// Let A_is be a program evaluator for E_i ▷ V ∪ {x_1, ..., x_n} → ∅,  1 ≤ i ≤ n
// Let A'_s be a program evaluator for E' ▷ V ∪ {x_1, ..., x_n} → V'

A_s = let // program evaluator for
      E = letrec x_1 = E_1, ..., x_n = E_n in E' ▷ V → V'
    (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_is
    (get-val', is-done', remove', add-inst', upd-val') = A'_s
    letrec
        A_1s'' = (get-val_1'', is-done_1'', remove_1'', add-inst_1'', upd-val_1'')
        R_1'' = registration.new(V ∪ {arg}, Type(E))
        link_1 = fun (π, α)  α ⊎ [x_1/A_1s'', ..., x_n/A_ns'']  end
        get-val_1'' = fun (π, α : ass_{V∪{arg}})  get-val_1(π, link_1(π, α))  end
        get-upd-com_1'' = fun (π, α, c_1)  c_1  end
        is-done_1'' = fun (π, α)  is-done_1(π, link_1(π, α))  end
        remove_1'' = gen-remove(R_1'', remove_1, link_1)
        add-inst_1'' = gen-add-inst(R_1'', add-inst_1, link_1, get-upd-com_1'')
        upd-val_1'' = gen-upd-val(R_1'', upd-val_1, link_1, get-upd-com_1'', gen-ass-com(link_1))
        ...
        A_ns'' = (get-val_n'', is-done_n'', remove_n'', add-inst_n'', upd-val_n'')
        R_n'' = registration.new(V ∪ {arg}, Type(E))
        link_n = fun (π, α)  α ⊎ [x_1/A_1s'', ..., x_n/A_ns'']  end
        get-val_n'' = fun (π, α : ass_{V∪{arg}})  get-val_n(π, link_n(π, α))  end
        get-upd-com_n'' = fun (π, α, c_n)  c_n  end
        is-done_n'' = fun (π, α)  is-done_n(π, link_n(π, α))  end
        remove_n'' = gen-remove(R_n'', remove_n, link_n)
        add-inst_n'' = gen-add-inst(R_n'', add-inst_n, link_n, get-upd-com_n'')
        upd-val_n'' = gen-upd-val(R_n'', upd-val_n, link_n, get-upd-com_n'', gen-ass-com(link_n))
    in
    W = V ∪ V' ∪ {x_1, ..., x_n}
    R = registration.new(V ∪ V', Type(E))
    link = fun (π, α)  α ⊎ [x_1/A_1s'', ..., x_n/A_ns'']  end
    get-val = fun (π, α)  R(π, α)  end
    get-upd-com = fun (π, α, c')  c'  end
    is-done = fun (π, α)  is-done'(π, link(π, α))  end
    remove = gen-remove(R, remove', link)
    add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
    upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.33: A program evaluator for recursive lets on streams s.

```
// Let A_is be a program evaluator for E_i ▷ V → ∅, 1 ≤ i ≤ n

A_s = let // program evaluator for E = (E_1, ..., E_n) ▷ V → ∅
   (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_is ,  1 ≤ i ≤ n
   R = registration.new(V, TYPE(E))
   link_i = fun(π, α)  α  end for all 1 ≤ i ≤ n
   get-val = fun(π, α)  R(π, α)  end
   get-upd-com = fun(π, α, c_1, ..., c_n)  (c_1, ..., c_n)  end
   is-done = fun(π, α)  is-done_1(π, link_1(π, α)) ∧ ... ∧ is-done_n(π, link_n(π, α))  end
   remove = gen-remove(R, remove_1, ..., remove_n, link_1, ..., link_n)
   add-inst = gen-add-inst(R, add-inst_1, ..., add-inst_n, link_1, ..., link_n, get-upd-com)
   upd-val = gen-upd-val(R, upd-val_1, ..., upd-val_n, link_1, ..., link_n, get-upd-com,
                gen-ass-com(link_1), ..., gen-ass-com(link_n))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.34: A program evaluator for tuple constructions on streams s.

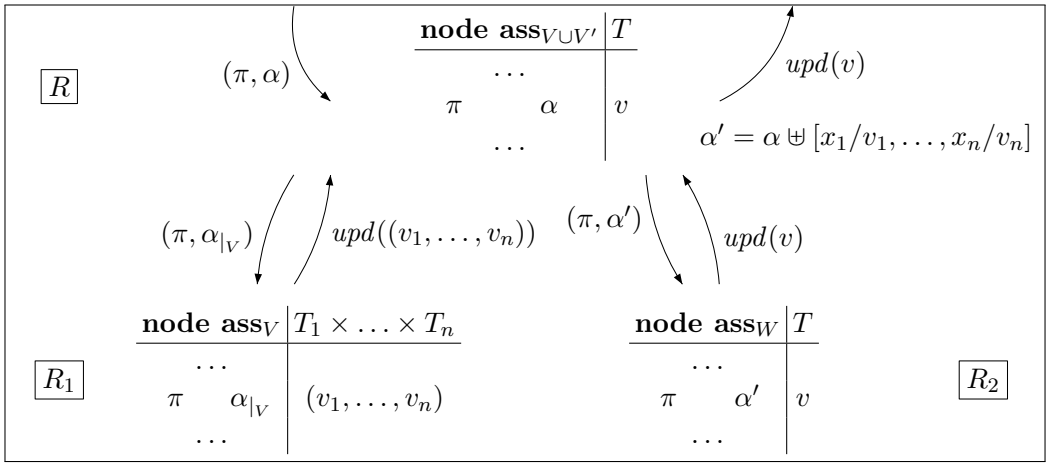


Tuples are constructed by program evaluator A_s, whereas program evaluator A_{is} computes the i-th component of the tuples. Therefore, at some event, the *get-upd-com* function of A_s combines the updates c_i to the i-th component into an update $(c_1, \ldots, c_n)$ of the tuple.

Tuple decompositions.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \cup \{x_1, \ldots, x_n\} \to V' \quad E = \textbf{match } E_1 \textbf{ with } (x_1, \ldots, x_n) \textbf{ in } E_2}{E \triangleright V \to V'}$$

Let A_{1s} be a program evaluator for $E_1 : T_1 \times \ldots \times T_n \triangleright V \to \emptyset$, and A_{2s} be a program evaluator for $E_2 : T \triangleright V \cup \{x_1, \ldots, x_n\} \to V'$. We define a program evaluator A_s for $E \triangleright V \to V'$ in Figure 11.35. Registrations have the following form where $W = V \cup V' \cup \{x_1, \ldots, x_n\}$:

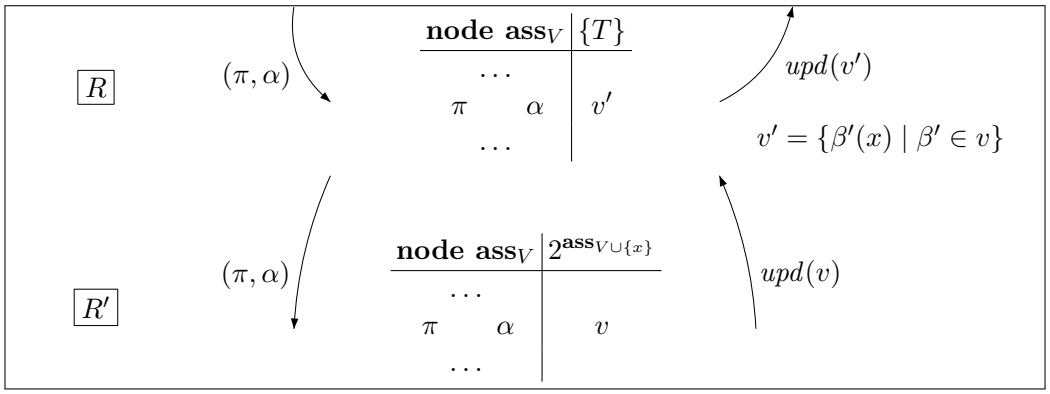


Program evaluator A_s decomposes tuples by A_{1s} in R_1 into their components v_i, which are passed to evaluator A_{2s} via assignments $\alpha' = \alpha \uplus [x_1/v_1, \ldots, x_n/v_n]$ with respect to the entries given for the registrations. Any updates of tuples in R_1 must therefore be forwarded as variable assignment update to A_{2s}. This is done by function *get-ass-com*$_2$, which given the updates U_1 by A_{1s} and given some variable assignment update *AssUps*, computes a new variable assignment update such that the *upd-val* of A_{2s} has knowledge of these updates. The function first converts the updates in *AssUps* to fit linked entries in R_2 and next adds for any update c of tuples $(v_1, \ldots, v_n)$ and any variable x_i a new variable assignment update that indicates that the value of variable x_i has been updated with the update to the i-th component of v_i. This update is computed via function *i-th-com* that decomposes the tuple update c into updates c_i for component i. The query evaluator B_s for $E \triangleright V \to V'$ differs with respect to the program evaluator A_s only in the link definition.

Set constructions.

$$\frac{E' \triangleright V \to \{x\} \quad E = \{x \mid E'\}}{E \triangleright V \to \emptyset}$$

Let B'_s be a query evaluator for $E' \triangleright V \to \{x\}$. We define program evaluator A_s for $E : \{T\} \triangleright V \to \emptyset$ in Figure 11.36. The registrations have the following form:



```
// Let A_{1s} be a program evaluator for E_1 ▷ V → ∅
// Let A_{2s} be a program evaluator for E_2 ▷ V ∪ {x_1, ..., x_n} → V'
// Let E = match E_1 with (x_1, ..., x_n) in E_2

A_s = let // program evaluator for E ▷ V → V'
  (get-val_1, is-done_1, remove_1, add-inst_1, upd-val_1) = A_{1s}
  (get-val_2, is-done_2, remove_2, add-inst_2, upd-val_2) = A_{2s}
  W = V ∪ V' ∪ {x_1, ..., x_n}
  R = registration.new(V ∪ V', TYPE(E))
  link_1 = fun(π, α) α_{|V} end
  link_2 = fun(π, α)
      case get-val_1(π, link_1(π, α))
      of • then α ⊎ [x_1/•, ..., x_n/•]
      of (v_1, ..., v_n) then α ⊎ [x_1/v_1, ..., x_n/v_n]
      end
  end
  get-val = fun(π, α) R(π, α) end
  get-upd-com = fun(π, α, c_1, c_2) c_2 end
  letrec
   i-th-com = fun(c, i)
       case c
       of none then none
       of c_1&c_2 then i-th-com(c_1, i)&i-th-com(c_2, i)
       of (c_1, ..., c_n) then c_i
   end
  in
  get-ass-com_2 = fun(AssUps, U_1)
      M = set.new()
      get-ass-com'_2 = gen-ass-com(link_2)
      M.union(get-ass-com'_2(AssUps, U_1))
      inv-links_1 = gen-inv-links(R, link_1)
      for (π, α_1, (v_1, ..., v_n), c) ∈ U_1 do
         for (π, α) ∈ inv-links_1(π, α_1) do
            M.add(π, α ⊎ [x_1/v_1, ..., x_n/v_n], x_1, i-th-com(c, 1))
            ...
            M.add(π, α ⊎ [x_1/v_1, ..., x_n/v_n], x_n, i-th-com(c, n))
   in M.content()
  end
  is-done = fun(π, α) is-done_1(π, link_1(π, α)) ∧ is-done_2(π, link_2(π, α)) end
  remove = gen-remove(R, remove_1, remove_2, link_1, link_2)
  add-inst = gen-add-inst(R, add-inst_1, add-inst_2, link_1, link_2, get-upd-com)
  upd-val = gen-upd-val(R, upd-val_1, link_1, upd-val_2, link_2, get-upd-com,
                gen-ass-com(link_1), get-ass-com_2)
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.35: A program evaluator for tuple decompositions on streams s.

```
// Let B'_s be a query evaluator for E' ▷ V → {x}

A_s = let // program evaluator for E = {x | E'} ▷ V → ∅
  (get-val', is-done', remove', add-inst', upd-val') = B'_s
  R = registration.new(V, TYPE(E))
  link = fun(π, α) α end
  get-val = fun(π, α) R(π, α) end
  letrec
   get-upd-var-ass-elem =
    fun(e, c)
      case c
      of none then none
      of c1&c2 then get-upd-var-ass-elem(e, c1)&get-upd-var-ass-elem(e, c2)
      of extend-var-ass(y, v) then if y = x then extend-set({v}) else none
      of upd-var-ass(y, v, c)) then if y = x then upd-elem-set(v, c) else none
    end
   get-upd-com =
    fun(π, α, c)
      case c
      of none then none
      of c1&c2 then get-upd-com(π, α, c1)&get-upd-com(π, α, c2)
      of extend-set(v) then extend-set({β'(x) | β' ∈ v, x ∈ β'.dom()})
      of upd-elem-set(e, c') then get-upd-var-ass-elem(e, c')
      of resolve-elem-set(e, b) then if x ∈ e.dom() then resolve-elem-set(e(x), b)
          else none
    end
  in
  is-done = fun(π, α) is-done'(π, α) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

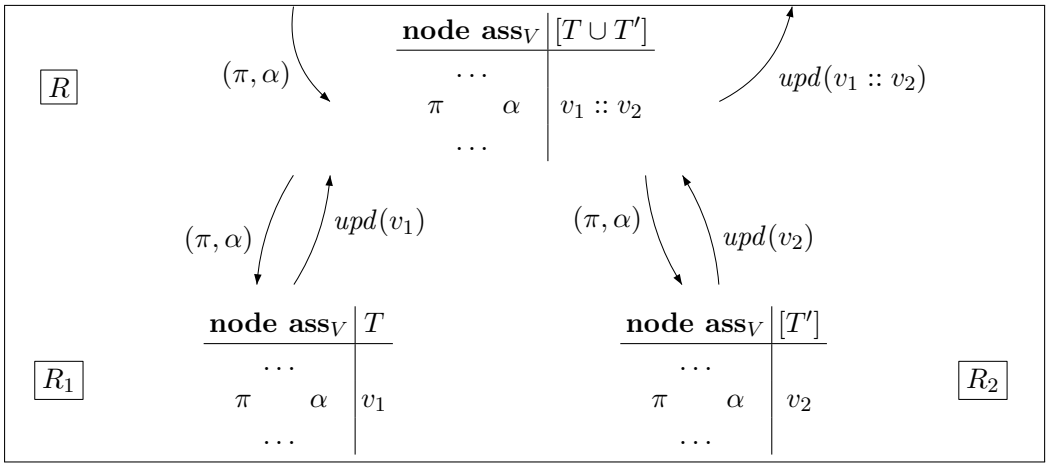
Figure 11.36: A program evaluator for set constructions on streams s.

The set of all elements of values for x is constructed via query evaluator B'_s that generates all possible values for x, by computing variable assignment extensions. Given an entry $(\pi, \alpha) \in R.dom()$ the value v of the linked entry $(\pi, \alpha) \in R'.dom()$ contains variable assignment that extend α by assigning values to x. The result set by A_s is thus $v' = \{\beta'(x) \mid \beta' \in v\}$. The challenge for the program evaluator A_s is thus to compute updates to v' given updates to v. The *get-upd-com* function therefore decomposes updates c to v. When v is extended by a set of variables assignments, then one simply extends v' by the set of values of x of these variable assignments. Whenever a variable assignment e of v that assigns some value to x is resolved by boolean b then one resolves element $e(x)$ in v' by b. Finally when a variable assignment e is updated via some variable assignment update c', then one calls function *get-upd-var-ass-elem* that returns an update to v that corresponds to c' when c' involves updates to values of x.

List constructions.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset \quad E = E_1 :: E_2}{E \triangleright V \to \emptyset}$$

Let A_{1s} be program evaluator for $E_1 : T \triangleright V \to \emptyset$ and A_{2s} be program evaluator for $E_2 : [T'] \triangleright V \to \emptyset$ We define program evaluator A_s for $E_1 :: E_2 : [T \cup T'] \triangleright V \to \emptyset$ in Figure 11.37. Registrations have the following form:



Lists are constructed by joining some head element that is computed by program evaluator A_{1s}, with a tail that is the list, that is computed by program evaluator A_{2s}. In function *get-upd-com* one translates updates c_1 to the head and updates c_2 to the tail to updates for the result list as follows. Updates c_1 become updates to the result list that update the first element in the list. Updates c_2 to the tail must be altered to updates to the result list, that contains a new head element, which is done via function *add-index* that converts updates to the i-th element of the tail, to updates to the $i+1$-st element of the result list.

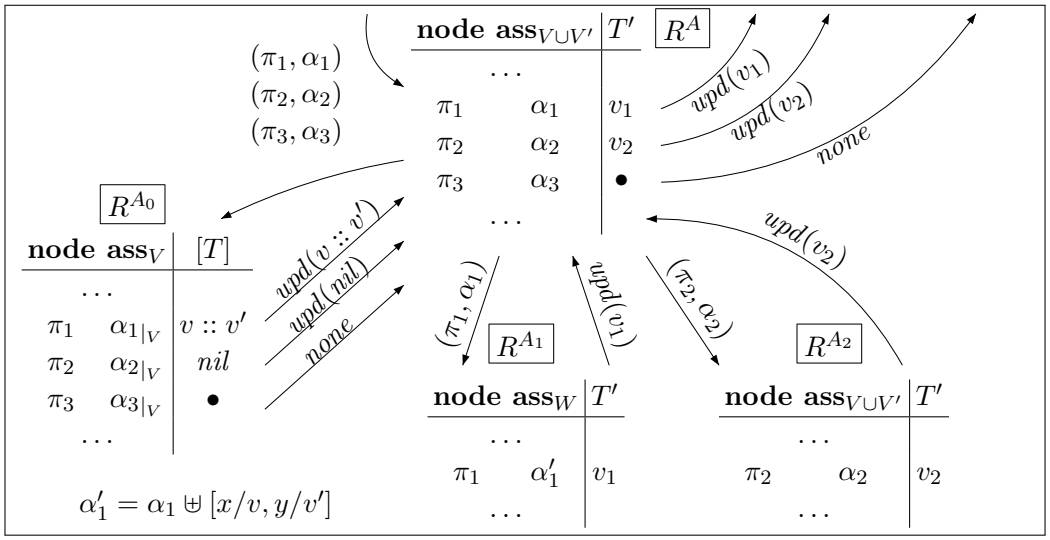
List decompositions.

$$\frac{\begin{array}{c} E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \cup \{x, y\} \to V' \quad E_3 \triangleright V \to V' \\ E = \textbf{match } E_0 \textbf{ with } x :: y \textbf{ then } E_1 \textbf{ else } E_2 \end{array}}{E \triangleright V \to V'}$$

Let A_{0s} be a program evaluator for $E_0 : [T] \triangleright V \to \emptyset$, A_{1s} a program evaluator for $E_1 : T' \triangleright V \cup \{x, y\} \to V'$, and A_{2s} a program evaluator for $E_2 : T' \triangleright V \to V'$. We define program evaluator A_s for $E : T' \triangleright V \to V'$ in Figure 11.38. Registrations have the following form, where $W = V \cup V' \cup \{x, y\}$:

```
// Let A_is be program evaluators for E_i ▷ V → ∅ and all 1 ≤ i ≤ 2

A_s = let // program evaluator for E = E_1 :: E_2 ▷ V → ∅
   (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_is for all 1 ≤ i ≤ 2
   R = registration.new(V, TYPE(E))
   link_1 = fun(π, α) α end
   link_2 = fun(π, α) α end
   get-val = fun(π, α) R(π, α) end
   letrec add-index = fun(π, α, c)
      case c
      of none then none
      of c_1&c_2 then add-index(π, α, c_1)&add-index(π, α, c_2)
      of extend-list(v) then extend-list(v)
      of upd-elem-list(i, c') then upd-elem-list(i + 1, c')
      of resolve-elem-list(i, b) then resolve-elem-list(i + 1, b)
    end
   in
   get-upd-com = fun(π, α, c_1, c_2) upd-elem-list(1, c_1)&add-index(π, α, c_2) end
   is-done = fun(π, α) is-done_1(π, link_1(π, α)) ∧ is-done_2(π, link_2(π, α)) end
   remove = gen-remove(R, remove_1, remove_2, link_1, link_2)
   add-inst = gen-add-inst(R, add-inst_1, add-inst_2, link_1, link_2, get-upd-com)
   upd-val = gen-upd-val(R, upd-val_1, upd-val_2, link_1, link_2, get-upd-com, gen-ass-com(link_1),
                gen-ass-com(link_2))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.37: A program evaluator for list constructions on streams s.



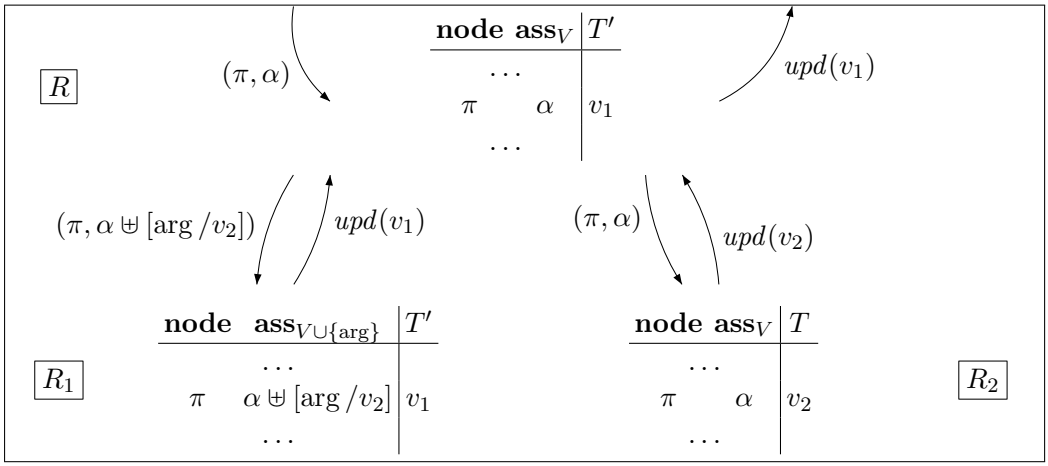
For algorithm A in the beginning it is not known a priori whether A_{0s} evaluates to the empty list, for which the result by A_{2s} is the answer, or whether A_{0s} evaluates to a non-empty list, for which the result by A_{1s} is needed. One therefore waits until the computation of the list by A_{0s} has finished, at which event we launch A_{1s} or A_{2s}, respectively. This is done by function *get-upd-com*, similar to conditionals. Due to our restrictions on list decompositions the possible late launch of either

A_{1s} or A_{2s} does not require information from the past of the stream. The query evaluator B_s is similar and differs only slightly in link definitions.

Function applications.

$$\frac{E_1 \triangleright V \to \emptyset \quad E_2 \triangleright V \to \emptyset \quad E = E_1 E_2}{E \triangleright V \to \emptyset}$$

Let A_{1s} be a program evaluator for $E_1 : T \to T' \triangleright V \to \emptyset$ and A_{2s} be a program evaluator for $E_2 : T \triangleright V \to \emptyset$. We define a program evaluator A_s for $E_1 E_2 : T' \triangleright V \to \emptyset$ in Figure 11.39. Registrations have the following form:



The argument v_2 that is computed by A_{2s} is passed via variable assignment $[\text{arg}\,/v_2]$ to A_{1s}. Any update c_1 to the result by v_1 by A_{1s} is therefore an update to the result v_1 of A_s of the function application, see function *get-upd-com*. However, similar to tuple decompositions, one requires to pass updates to argument v_2 as variable assignment updates to A_{2s}, computed by function *get-ass-com*$_1$. Then after the registration R_2 has been updated, one can update R_1, as one sees in the order of the parameters for the function generator *gen-upd-val*.

Function definitions.

$$\frac{E' \triangleright V \cup \{x\} \to \emptyset \quad E = \lambda x.E'}{E \triangleright V \to \emptyset}$$

Let A'_s be a program evaluator for $E' : T' \triangleright V \cup \{x\} \to \emptyset$. We define a program evaluator A_s for applying function definitions $\lambda x.E' : T \to T' \triangleright V \to \emptyset$ in Figure 11.40. Registrations have the following form:

```
// Let A_{0_s} be a program evaluator for E_0 ▷ V → ∅
// Let A_{1_s} be a program evaluator for E_1 ▷ V ∪ {x, y} → V'
// Let A_{2_s} be a program evaluator for E_2 ▷ V → V'
// Let E = match E_0 with x :: y then E_1 else E_2

A_s = let // program evaluator for E ▷ V → V'
  (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_{i_s} for all 0 ≤ i ≤ 2
  W = V ∪ V' ∪ {x, y}
  R = registration.new(V ∪ V', TYPE(E))
  link_0 = fun(π, α) α_{|V} end
  link_1 = fun(π, α)
      case get-val_0(π, link_0(π, α))
      of • then α ⊎ [x/•, y/•]
      of v :: v' then α ⊎ [x/v, y/v']
      end
  end
  link_2 = fun(π, α) α end
  get-val = fun(π, α) R(π, α) end
  get-upd-com = fun(π, α, c_0, c_1, c_2)
      if c_0 ≠ none ∧ is-done_0(π, link_0(π, α)) then
        if get-val_0(π, link_0(π, α)) = nil then add-inst_2(π, link_2(π, α))
        else add-inst_1(π, link_1(π, α))
      else
        if none then
          if is-done_0(π, link_0(π, α)) then
            case get-val_0(π, link_0(π, α))
            of nil then c_2
            of v :: v' then c_1
            end
          else none
        else none
  end
  is-done = fun(π, α) (get-val_0(π, α_{|V}) = true ∧ is-done_1(π, link_1(π, α)))
      ∨(get-val_0(π, α_{|V}) = false ∧ is-done_2(π, link_2(π, α))) end
  remove = gen-remove(R, remove_0, remove_1, remove_2, link_0, link_1, link_2)
  add-inst = fun(π, α)
      R.add-inst(π, α)
      c_0 = add-inst_0(π, link_0(π, α))
      c = get-upd-com(π, α, c_0, none, none)
   in c
  end
  upd-val = gen-upd-val(R, upd-val_0, upd-val_1, upd-val_2, link_0, link_1, link_2, get-upd-com,
              gen-ass-com(link_0), gen-ass-com(link_1), gen-ass-com(link_2))
in (get-val, is-done, remove, add-inst, upd-val)
```

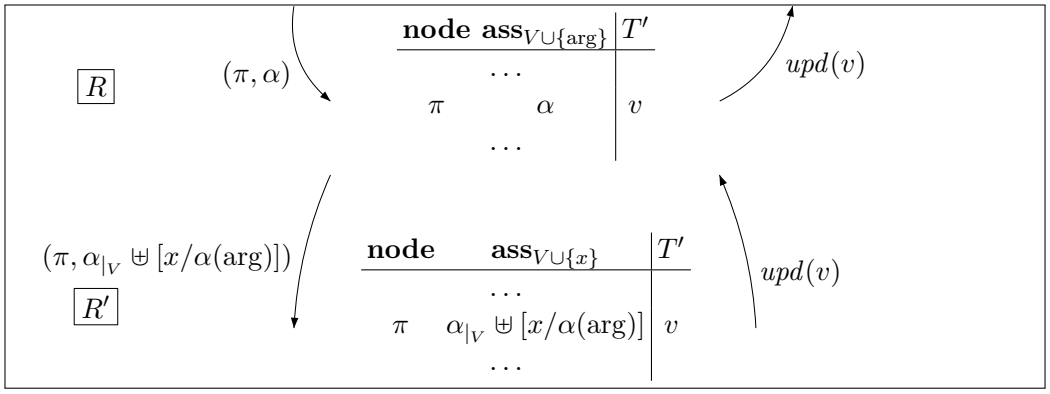
Figure 11.38: A program evaluator for list decompositions on streams s.

```
// Let A_is be a program evaluator for E_i ▷ V → ∅, 1 ≤ i ≤ 2
// Let arg be a fixed variable used to pass arguments

A_s = let // program evaluator for E = E_1 E_2 ▷ V → ∅
  (get-val_i, is-done_i, remove_i, add-inst_i, upd-val_i) = A_is , 1 ≤ i ≤ 2
  R = registration.new(V, TYPE(E))
  link_1 = fun(π, α) α ⊎ [arg/get-val_2(π, α)] end
  link_2 = fun(π, α) α end
  get-val = fun(π, α) R(π, α) end
  get-upd-com = fun(π, α, c_1, c_2) c_1 end
  get-ass-com_1 = fun(AssUps, U_2)
      M = set.new()
      get-ass-com'_1 = gen-ass-com(link_1)
      M.union(get-ass-com'_1(AssUps, U_2))
      inv-links_2 = gen-inv-links(R, link_2)
      for (π, α_2, v, c) ∈ U_2 do
        for (π, α) ∈ inv-links_2(π, α_2) do
          M.add(π, α ⊎ [arg/v], arg, c)
    in
      M.content()
  end
  is-done = fun(π, α) is-done_1(π, link_1(π, α)) ∧ is-done_2(π, link_2(π, link_2(π, α))) end
  remove = gen-remove(R, remove_1, remove_2, link_1, link_2)
  add-inst = gen-add-inst(R, add-inst_1, add-inst_2, link_1, link_2, get-upd-com)
  upd-val = gen-upd-val(R, upd-val_2, upd-val_1, link_2, link_1, get-upd-com,
              gen-ass-com(link_2), get-ass-com_1)
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.39: A program evaluator for function applications for streams s.

```
// Let A'_s be a program evaluator for E' ▷ V ∪ {x} → ∅

A_s = let // program evaluator for E = λx.E' ▷ V → ∅
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   R = registration.new(V ∪ {arg}, T')
   link = fun(π, α) α|_V ⊎ [x/α(arg)] end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c) c end
   is-done = fun(π, α) is-done'(π, link(π, α)) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.40: Program evaluator for function definitions for streams s.

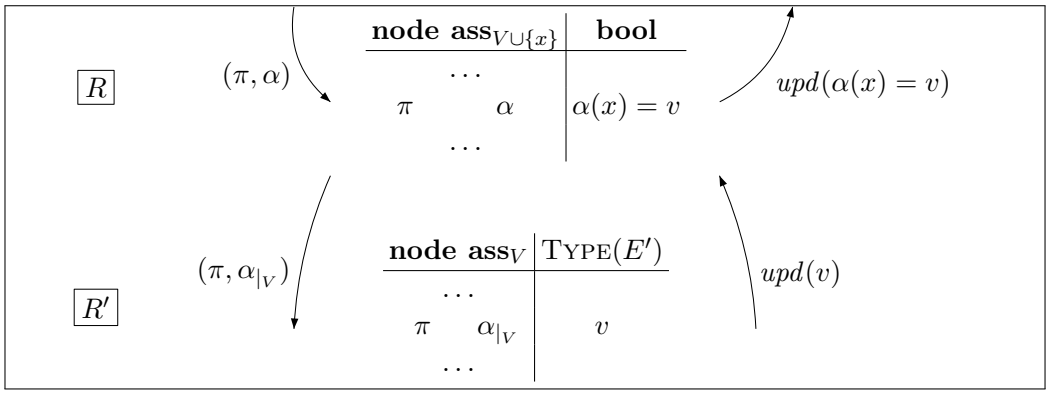


A_s only applies the function definition to some argument. The evaluation of the function body by A'_s therefore computes the required result.

Variable Bindings.

$$\frac{E' \triangleright V \to \emptyset \quad E = (x = E')}{E \triangleright V \to \{x\}}$$

Let A'_s be a program evaluator for $E' \triangleright V \to \emptyset$. We define program evaluator A_s for $x = E' \triangleright V \to \{x\}$ in Figure 11.41. Registrations have the following form:

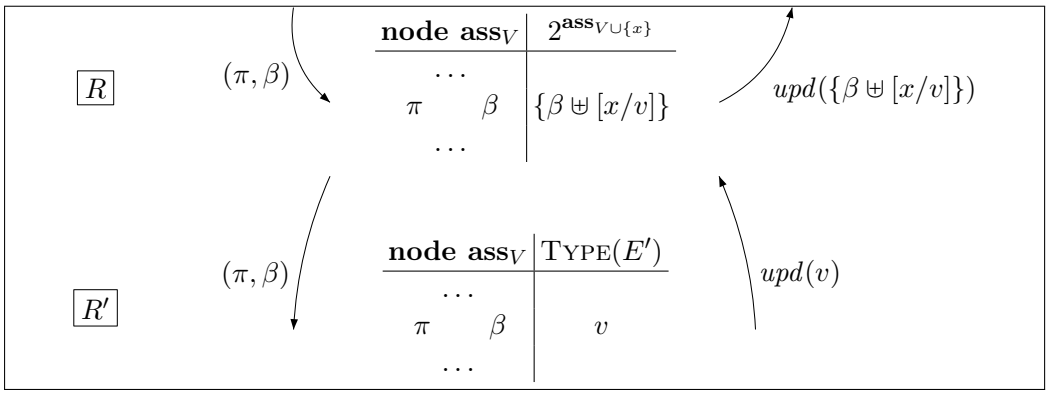


```
// Let A'_s be a program evaluator for E' ▷ V → ∅

A_s = let // program evaluator for E = (x = E') ▷ V → {x}
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   R = registration.new(V ∪ {x}, TYPE(E))
   link = fun(π, α) α|_V end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c) if c ≠ none ∧ is-done(π, α) then
        instantiate(α(x) = get-val'(π, link(π, α))) else none end
   is-done = fun(π, α) is-done'(π, α) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.41: A program evaluator for variable bindings on streams s.

Since a boolean is computed by A_s, one waits until the answer by A'_s for some linked entry is determined. The *get-upd-com* function then returns the update of the truth value of whether the answer v by A'_s is equal to value of x in the variable assignment α of the respective input entry.

Let A'_s be a program evaluator for $E' \triangleright V \to \emptyset$. We define query evaluator B_s for $x = E' \triangleright V \to \{x\}$ in Figure 11.42. Registrations have the following form:



Query evaluator B_s must generate the value v for x that A_s computes, and must thus return the set that contains the variable assignment of any input entry extended by $[x/v]$. Given some update c to some result value v by A_s, the *get-upd-com* function returns an update to the value that is assigned to variable x in the variable assignment of the answer set.

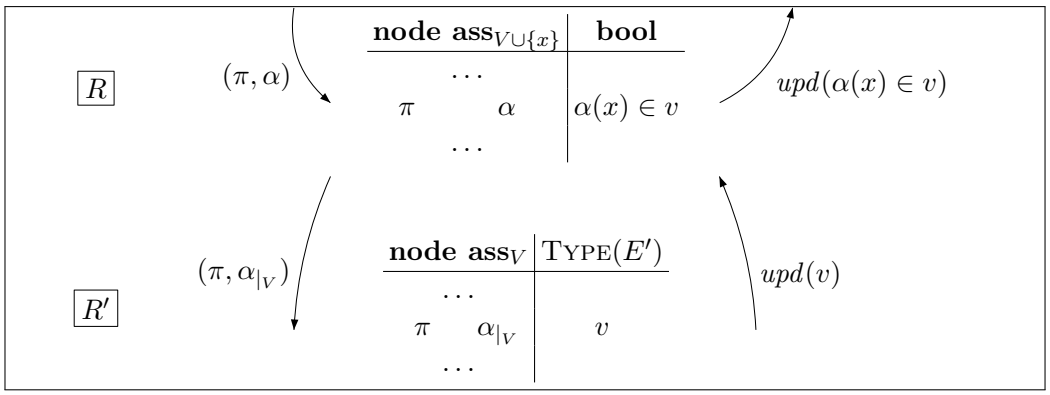
Set Membership.

$$\frac{E' \triangleright V \to \emptyset \quad E = (x \in E')}{E \triangleright V \to \{x\}}$$

```
// Let A'_s be a program evaluator for E' ▷ V → ∅

B_s = let // query evaluator for E = (x = E') ▷ V → {x}
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   R = registration.new(V, TYPE(E))
   link = fun(π, β) β end
   get-val = fun(π, β) R(π, β) end
   get-upd-com = fun(π, β, c)
      {v} = get-val(π, β)
      upd-elem-set(v, upd-var-ass(x, v(x), c)
   end
   is-done = fun(π, β) is-done'(π, β) end
   remove = gen-remove(R, remove', link)
   add-inst = fun(π, β)
       R.add-inst(π, β)
       c' = add-inst(π, link(π, β))
       v = {β ⊎ [x/get-val'(π, link(π, β))]}
       R.set-val(π, β, v)
    in
       extend-set(v)
   end
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.42: A query evaluator for variable bindings for streams s.

Let A'_s be a program evaluator for $E' \triangleright V \to \emptyset$. We define program evaluator A_s for $x \in E' \triangleright V \to \{x\}$ in Figure 11.43. Registration have the following form:

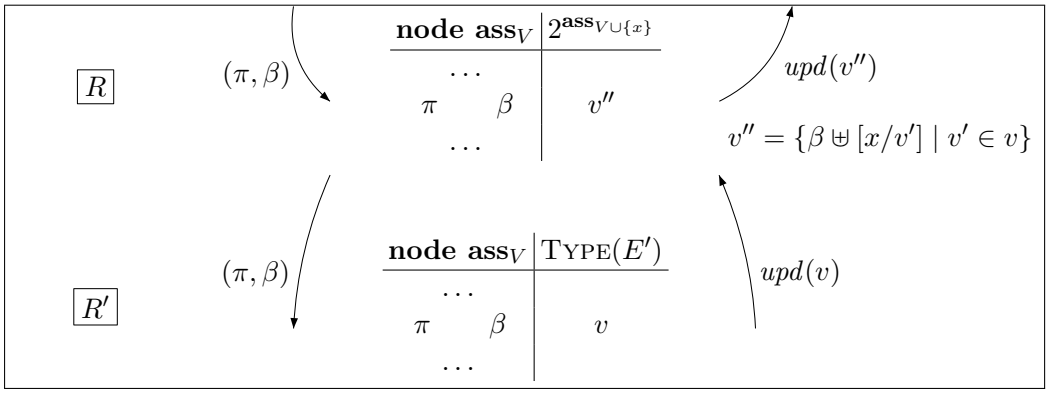


Similar to the program evaluator for variable bindings, the program evaluator A_s for the set membership waits until the set v by A'_s is determined, such that *get-upd-com* returns the update that instantiates to the respective membership test.

Let A'_s be a program evaluator for $E' \triangleright V \to \emptyset$. We define query evaluator B_s for $x \in E' \triangleright V \to \{x\}$ in Figure 11.44. Registrations have the following form:

```
// Let A'_s be a program evaluator for E' ▷ V → ∅

A_s = let // program evaluator for E = (x ∈ E') ▷ V → {x}
   (get-val', is-done', remove', add-inst', upd-val') = A'_s
   R = registration.new(V ∪ {x}, TYPE(E))
   link = fun(π, α) α|_V end
   get-val = fun(π, α) R(π, α) end
   get-upd-com = fun(π, α, c) if c ≠ none ∧ is-done(π, α) then
        instantiate(α(x) ∈ get-val'(π, link(π, α))) else none end
   is-done = fun(π, α) is-done'(π, α) end
   remove = gen-remove(R, remove', link)
   add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
   upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.43: A program evaluator for set membership testing on streams s.



Query evaluator B_s must generate all values for x that arise from the set v that A'_s computes. Given some entry $(\pi, \beta) \in R.dom()$ and some set v by A'_s for some linked entry, B_s computes the set of variable assignments $\{\beta \uplus [x/v'] \mid v' \in v\}$. The *get-upd-com* function thus turns an update c to v into a corresponding update for the result by B_s.

11.3 Examples

11.3.1 Navigational Queries

In the following we present the evaluation of a forward and a backward query, to show the control of tuple generation and the interaction with registration tables that may select nodes from the past.

A Forward Query. Let $F = ch^*(a \wedge is_x)$ be a forward query of interest with mode $F \triangleright \emptyset \to \{x\}$ that selects all a-descendants. We thus consider

$$start(F, z) = ch^*(is_z \wedge ch^*(a \wedge is_x)).$$

```
// Let A'_s be a, program evaluator for E' ▷ V → ∅

B_s = let // query evaluator for E = (x ∈ E') ▷ V → {x}
  (get-val', is-done', remove', add-inst', upd-val') = A'_s
  R = registration.new(V, TYPE(E))
  link = fun(π, β) β end
  get-val = fun(π, β) R(π, β) end
  letrec
   get-upd-com =
    fun(π, β, c)
      case c
      of none then none
      of c1&c2 then get-upd-com(π, β, c1)&get-upd-com(π, β, c2)
      of extend-set(v) then extend-set({β ⊎ [x/v'] | v' ∈ v})
      of upd-elem-set(e, c') then upd-elem-set(β ⊎ [x/e], upd-var-ass(x, e, c'))}
      of resolve-elem-set(e, b) then resolve-elem-set(β ⊎ [x/e], b)
    end
  in
  is-done = fun(π, β) is-done'(π, β) end
  remove = gen-remove(R, remove', link)
  add-inst = gen-add-inst(R, add-inst', link, get-upd-com)
  upd-val = gen-upd-val(R, upd-val', link, get-upd-com, gen-ass-com(link))
in (get-val, is-done, remove, add-inst, upd-val)
```

Figure 11.44: A query evaluator for set membership testing on streams s.

The query contains no backward axes, such that we can directly translate it to an ENWA A. The alphabet of A is annotated by subsets of $\mathcal{V}(start(F, z)) = \{z, x\}$, i.e. the variables contained in $start(F, z)$, and thus we are generating pairs of nodes. Our example in Figure 11.45 evaluates A over a prefix of the linearization of tree $t = a(b(a))$. The registration contains variables z, x, and *conf* for configurations of A. We surrounded the second event by a rectangle to indicate that we want to evaluated F at the second node π_2 of the tree. Before evaluation starts, there is an initial entry contained in the registration where c_0 is the initial configuration of A in an initial state with the empty stack, and for variables z and x there is a $\bullet$ entry, for future nodes. The automaton will try to instantiate future $\bullet$ values by the node of open events, for which it does not reject. However, it may only instantiate the $\bullet$ for variable z if we instruct it to do so, thus at the second event. After reading event (op, π_1) of the first node π_1, the configuration of the automaton for the begin-candidates changes to c_1, since the stack is updated. The intuitive meaning of a pair (v_1, v_2) of values for variables z and x, respectively, with respect to the query, is that v_2 must be an a-descendant of v_1. At the first event we did not instantiate the z-$\bullet$ value with π_1, since we did not start the evaluation of F at that node, and the automaton could not instantiate the $\bullet$-value for x, since a pair $(\bullet, \pi_1)$ would intuitively mean that π_1 is an a-descendant of some future node, which is forbidden according to the automaton. At the next event (op, π_2) we want to evaluate F, and thus instruct the automaton to instantiate node π_2 for variable z. At the next event

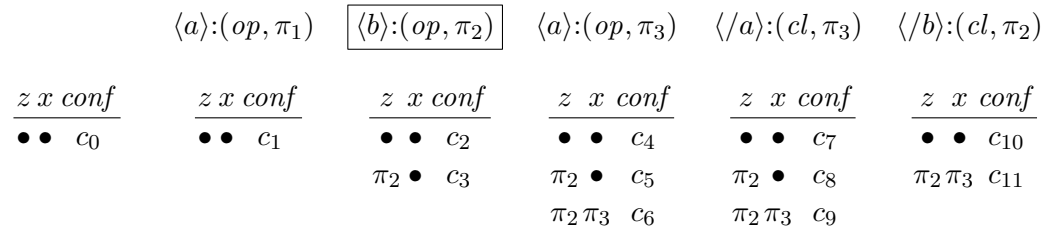


Figure 11.45: Content of the registration table for the evaluation of query $ch^*(is_z \wedge ch^*(a \wedge is_x))$ over the linearization of tree $a(b(a))$ until event $\langle/b\rangle$.

(op, π_3) the automaton instantiates the x-• value of pair $(\pi_2, \bullet)$ with π_3 and moves into a selecting configuration c_6 with a selection state. Pair (π_2, π_3) is an answer: node π_3 is an a-descendant of node π_2. At event (cl, π_2) where we close node π_2, notice that the automaton removes pair $(\pi_2, \bullet)$: indeed it is not possible that node π_2 finds further a-descendants.

A Backward Query. Queries that contain backward axes are evaluated just as similar. Consider query $F = par^+(is_x)$ with mode $F\triangleright \to \{x\}$ that selects all ancestors x of a given start node. Backward axes are eliminated in $start(F, z) = ch^*(is_z \wedge par^+(is_x))$. The backward axes free query is

$$is_x \wedge ch^+(is_z) \vee ch^+(is_x \wedge ch^+(is_z))$$

which translates to some ENWA. The evaluation of this query over the same tree $t = a(b(a))$ as before is shown in Figure 11.46. There at the first event the registration contains the pair $(\bullet, \pi_1)$, which means that node π_1 is an ancestor of any future node to come. In general, our algorithm pre-computes all nodes selected by some query that may contain backward axes when started at future nodes. Also notice, that we precompute only those nodes that are candidates according to our ENWA: The speculative computation therefore correlates to the earliest-ness of our automaton. Finishing with the example, one sees at the next event (op, π_2) that node π_2 is another ancestor of future nodes, and also since we instructed an evaluation of the query at this node, we instantiated the • in pair $(\bullet, \pi_1)$ with node π_2 and obtain an answer pair (π_2, π_1) that is stored: Node π_2 has node π_1 as ancestor.

11.3.2 A Network of Queries

We consider query

$$F = \{x \mid par^+(is_x) \wedge lab_{name}(x) = \text{“a”}\}$$

that may be evaluated relatively to some node π. It returns the set of a-ancestor nodes of π. In particular, we are interested in subformula

$$lab_{name}(x) = \text{“a”}.$$

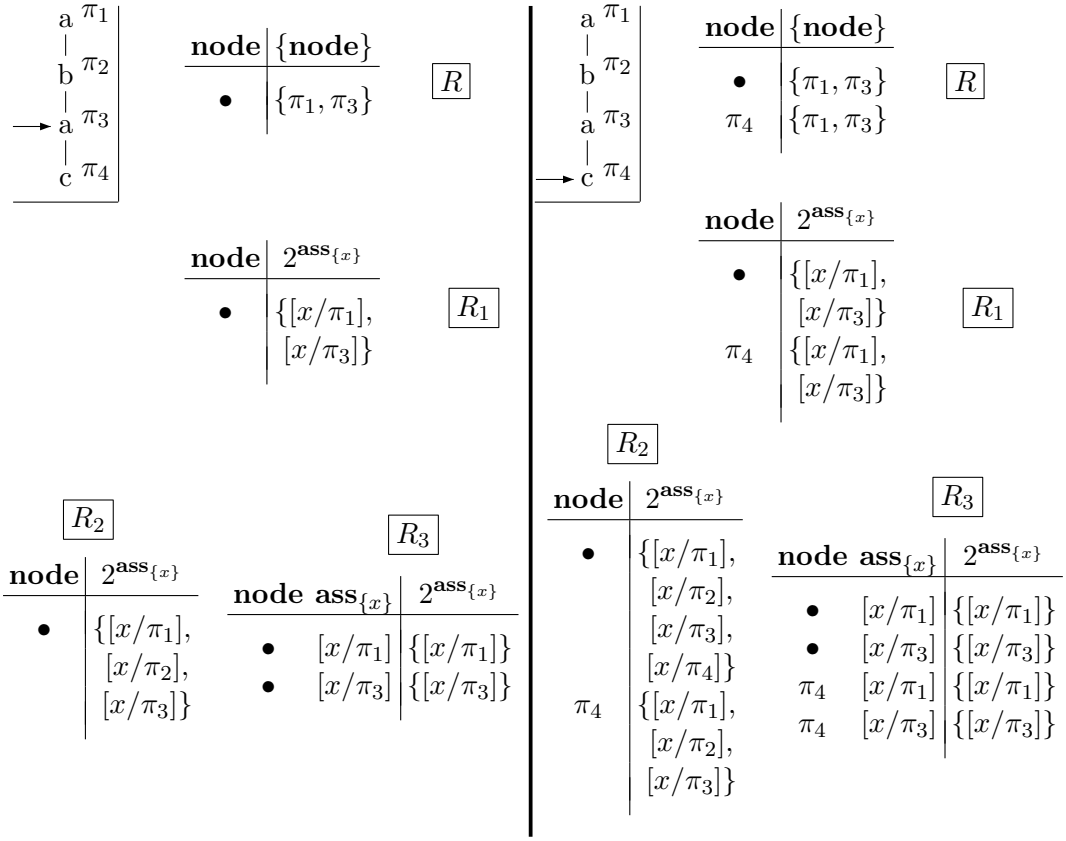
	$\langle a\rangle{:}(op,\pi_1)$	$\boxed{\langle b\rangle{:}(op,\pi_2)}$	$\langle a\rangle{:}(op,\pi_3)$	$\langle/a\rangle{:}(cl,\pi_3)$	$\langle/b\rangle{:}(cl,\pi_2)$
$z\ x\ conf$	$z\ x\ conf$	$z\ x\ conf$	$z\ x\ conf$	$z\ x\ conf$	$z\ x\ conf$
$\bullet\ \bullet\ c_0$	$\bullet\ \bullet\ c_1$	$\bullet\ \bullet\ c_3$	$\bullet\ \bullet\ c_7$	$\bullet\ \bullet\ c_{12}$	$\bullet\ \bullet\ c_{16}$
	$\bullet\ \pi_1\ c_2$	$\bullet\ \pi_1\ c_4$	$\bullet\ \pi_1\ c_8$	$\bullet\ \pi_1\ c_{13}$	$\bullet\ \pi_1\ c_{17}$
		$\bullet\ \pi_2\ c_5$	$\bullet\ \pi_2\ c_9$	$\bullet\ \pi_2\ c_{14}$	$\pi_2\,\pi_1\ c_{18}$
		$\pi_2\,\pi_1\ c_6$	$\bullet\ \pi_3\ c_{10}$	$\pi_2\,\pi_1\ c_{15}$	
			$\pi_2\,\pi_1\ c_{11}$		

Figure 11.46: Content of the registration table for the evaluation of query $ch^*(is_z \wedge par^+(is_x))$ over the linearization of tree $a(b(a))$ until event $\langle/b\rangle$.

of F that has one free variable x, whose value may be a node that comes before a node for which the query is evaluated. We give in Figure 11.47 a decomposition of F and display the contents of registration table for program and query evaluators in the network that evaluate the query. We evaluate F on the linearization of tree $a(b(a(c)))$, while on the left in the figure we showcase the buffers after event (op, π_3) has arrived (indicated by an arrow pointing to the node in the tree in the upper left corner of the figure), while on the right in the figure we showcase buffers after processing event (op, π_4) at which we start an evaluation. At event (op, π_3), notice, similar to as we seen before, that the registration R_2 for $par^+(is_x)$ finds ancestors π_1, π_2, and π_3 for any future node to come. The registration R_1 for the conjunction according to the mode system instruct for every tuple $\tau \in R_2$ an evaluation of the algorithm for query $lab_{name}(x) =$ "a" as we see in registration R_3. The label test, however, is satisfied for nodes π_1 and π_3 only, for which only those corresponding tuples remain in R_3. The algorithm for the conjunction combines the answers and synchronizes the tuples of R_3 with its registration table R_1. The algorithm that computes the set of all x such that the conjunction holds, now has precomputed the open set of nodes π_1 and π_3 as an answer for any future node. At the next event, in the figure on the right, one instructs an evaluation at node π_4, for which $\bullet$ values for variable z are instantiated by π_4. One observes that in registration table R_3 there is an entry $(\pi_4, [x/\pi_1])$ and an entry $(\pi_4, [x/\pi_3])$ of a start node π_4 with values for variable x from the past. However, since we pre-computed the answers of the label test for past nodes, the answer is already computed, and one just takes it over, but leaves entries $(\bullet, [x/\pi_1])$ and $(\bullet, [x/\pi_3])$ within the registration table, for further future nodes.

11.4 Tuple Sharing

Tuple sharing lifts the stack-and-state sharing algorithm of Section 8.6.3 to tuple sharing in networks of early nested word automata. The need for tuple sharing arises when the number of candidates grows linearly with the size of the document. For example, XPath query `//a/fn:data(following::b)` that computes for every



$$
\begin{array}{lcll}
 & & \overbrace{}^{F_1} \wedge \overbrace{\phantom{lab_{name}(x) = \text{“a”}\}}}^{F_2} & \\
R & = & \{x \mid par^+(is_x) \wedge lab_{name}(x) = \text{“a”}\} & \triangleright\ \emptyset \to \emptyset \\
R_1 & = & F_1 \wedge F_2 & \triangleright\ \emptyset \to \{x\} \\
R_2 & = & F_1 & \triangleright\ \emptyset \to \{x\} \\
R_3 & = & F_2 & \triangleright\ \{x\} \to \emptyset
\end{array}
$$

Figure 11.47: A network.

a-descendant the typed value of all following b-nodes. On a document whose root contains a list of a-children followed by a list of b-children, without tuple sharing, the typed value for every b-node will be computed, but this for every buffered a-node.

The idea is that when the automaton evaluator with stack-and-state sharing reaches for several candidates the same state and the same stack representative (of which there are no more than states as shown by Proposition 9), then these candidates can be shared in the tables, so that the work for them will be shared from that event on. This happens in the above example, where for every a-node there is an entry in the registration table that computes subquery `following::b`, all of which have the same configuration. At the arrival of the first b-node, the registration that computes the typed value of the b-node only does it once thanks

```
FJ:  //people/person[not(@id = following::bidder/personref/@person)]
BJ:  //open_auction
     [not(@id = preceding::people/person/watches/watch/@open_auction)]
AFJ: //people/person[not(@id = //bidder/personref/@person)]
ABJ: //open_auction[not(@id = //people/person/watches/watch/@open_auction)]
```

Figure 11.48: Additional data comparison queries.

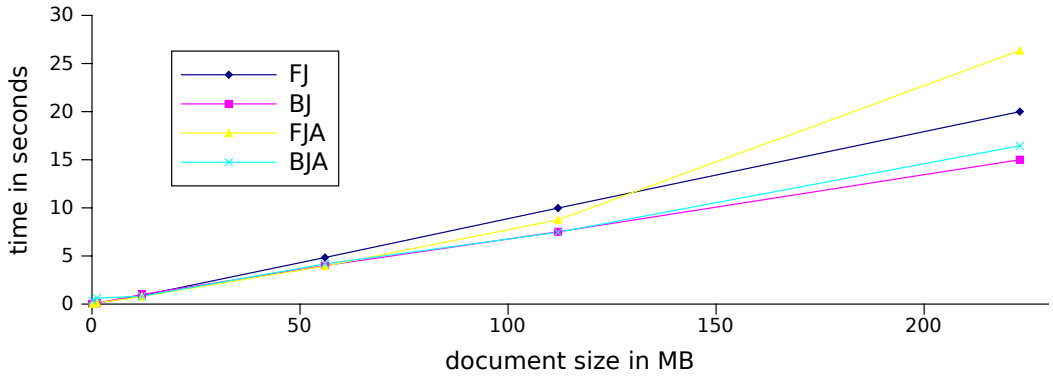


Figure 11.49: Parsing-free time of λXP for the queries of Figure 11.48.

to tuple sharing, instead of computing the same result for every a-node.

11.5 Experiments

The design of the λXP language is ahead of our current implementation in our QuiXPath tool. QuiXPath however implements besides navigational XPath, a large subset of the XPath functions, arithmetic, aggregation, and data comparisons in streaming. This implementation is based on the semantics and data model of XPath 1.0 and supports practically all XPath 1.0 expressions. The implementation enjoys linear runtime behavior for most aggregation and data comparison queries, thanks to tuple sharing.

In Figure 11.49 we showcase that QuiXPath runs indeed linearly for a collection of data comparison queries, that we list in Figure 11.48. They perform joins with forward and backward axes both with relative and absolute subqueries. There list of tables of data values grow largely over the stream of XPathMark documents, in contrast to the data comparison queries of the benchmark in Section A.3. A second experiment for the aggregation queries of Section A.4 of the XPathMark benchmark is done in Figure 11.50 scaling documents in size.

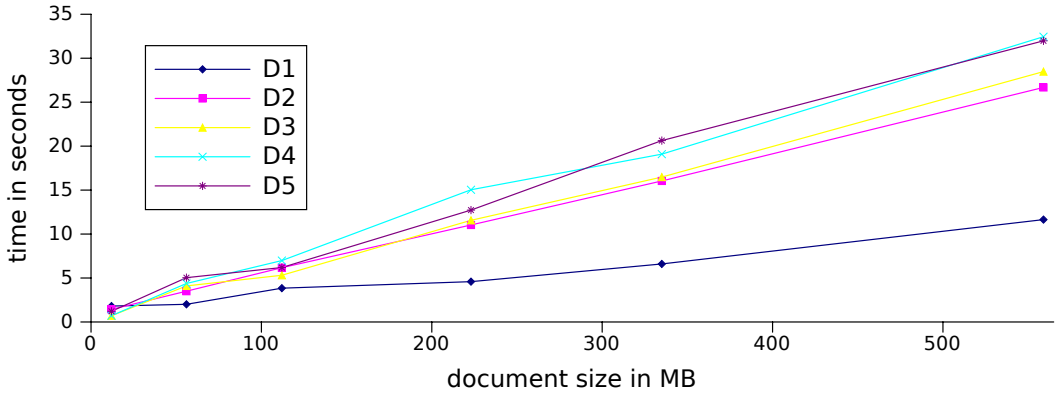


Figure 11.50: Parsing-free time of λXP for aggregation queries of Section A.4.

Chapter 12

Conclusion and Outlook

In this thesis, we have developed streaming algorithms for XPath with high time efficiency, low latency, low memory consumption, and large coverage.

The core of our algorithms maps navigational XPath queries to early nested word automata. The first idea here was to approximate earliest query answering by introducing selection and rejection states during the compilation, in order to avoid the need to decide accessibility problems for nested word automata. The second idea was to introduce a stack-and-state sharing algorithm. This was essential to avoid quadratic or higher blow-ups in the processing time for documents, in cases where the number of candidates is large. Rather than depending on the number of candidates, the time for processing a single event is reduced to depend only on the number states of the eNwa. We improved these algorithms further in time efficiency by developing projection algorithms for early nested word automata that permit to project large parts of the Xml streams away that the automata thereby does not have to process. We therefore obtained a streaming algorithm for answering navigational XPath queries with close to optimal low latency and memory consumption, whose implementation outperforms all other available tools in time efficiency.

Next, we worked on finding streaming algorithms that answer to general XPath 3.0 queries. For this purpose, we proposed λXP, a typed first-order logical language with higher order functions based on first principles for querying data trees. We then show that it is sufficiently expressive to capture all of XPath 3.0 by presenting a compiler from XPath 3.0 to λXP. This compiler provides a formal semantics to XPath 3.0 – for the first time to the best of our knowledge – i.e., it formalizes the W3C standard for XPath 3.0. We then propose a restriction of λXP that is accessible to streaming. The problem is to avoid exhaustive speculative computation. For instance, one might want to rule out cases, where some navigational queries are computed dynamically from the input document, and then applied to the same input document. Furthermore, exhaustive speculative computation bears the risk of running incorrectly into nontermination, when wrongly speculating that the result of a computation may be needed which is nonterminating. Finally, we developed streaming algorithms for λXP queries that are accessible to streaming. Our idea was to decompose general queries into network of simpler subqueries, so that our streaming algorithms for navigational subqueries can be lifted to the general case. To this purpose we proposed to decompose general queries into networks of registrations with eNwas. In order to avoid quadratic or higher blow-ups on the

network level, we propose to lift the stack-and-state sharing algorithm on the level of eNwAs to a tuple sharing algorithm on the level of registration networks. We implemented our algorithm to a large extend, and show that it covers very large fragments of XPath 3.0 in streaming mode. While mild extensions are still possible, it does not seem reasonable to us to try to process non-streaming-accessible queries of XPath 3.0 in streaming mode.

In the following, we envisage to complete our implementation of the presented streaming algorithms for the supported fragment of λXP. Then, they can serve as a streaming XPath motor for streaming implementations of Xslt, XQuery, and finally XProc. In [Labath 2015] the authors proposed a uniform programming language, similar to that of λXP, that allows to implement the W3C standards. This language extends on λXP by facilities to create and process data trees on the fly as one requires for transformation languages like Xslt and XQuery.

For the future, it will be relevant to develop streaming algorithms for NoSql databases. These rely on the Json format, whose data trees are then linearized to complex Json events. The challenge is to find streaming algorithms that can be combined with the algorithms of classical databases and that run in distributed settings and may thereby profit from parallelism.

Appendix A

XPathMark Benchmark

Contents

A common XPath benchmark is the XPathMark benchmark[1] which is the revised version in contrast to the benchmark of the original paper [Franceschet 2005]. It contains queries about XMark documents that contain a table of bids and a table of bidders. Some of the queries are parameterized by an integer $i \geq 1$, so that these queries can be scaled in size. Notice that the XPathMark bechmark was designed to test XPath 1.0 queries only, which are valid XPath 3.0 queries however. We state all of the queries of the benchmark, while we do not test against all of the queries in our experiments.

A.1 Downward Queries

Downward queries use forward axes child and descendant, path compositions, and filters.

A1: `/site/closed_auctions/closed_auction/annotation/description/text/keyword`
A2: `//closed_auction//keyword`
A3: `/site/closed_auctions/closed_auction//keyword`
A4: `/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date`
A5: `/site/closed_auctions/closed_auction[descendant::keyword]/date`
A6: `/site/people/person[profile/gender and profile/age]/name`
A7: `/site/people/person[phone or homepage]/name`
A8: `/site/people/person[address and (phone or homepage) and (creditcard or profile)]/name`

[1] http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html

A.2 Axis Queries

The axis queries in the benchmark use in addition to the downward axes, also backward axes, attribute axes and other forward axes. Some of the queries are parametric.

B1: `/site/regions/*/item[parent::namerica or parent::samerica]/name`

B2: `//keyword/ancestor::listitem/text/keyword`

B3: `/site/open_auctions/open_auction/bidder[following-sibling::bidder]`

B4: `/site/open_auctions/open_auction/bidder[preceding-sibling::bidder]`

B5: `/site/regions/*/item[following::item]/name`

B6: `/site/regions/*/item[preceding::item]/name`

B7: `//person[profile/@income]/name`

B8: `/site/open_auctions/open_auction[bidder and not(bidder/preceding-sibling::bidder)]/interval`

B9:
```
/site/open_auctions/open_auction
[(not(bidder/following::bidder) or
  not(bidder/preceding::bidder)) or
 (bidder/following::bidder and bidder/preceding::bidder)]
/interval
```

B10:
```
/site/open_auctions/open_auction
[(not(bidder/following::bidder) or
  not(bidder/preceding::bidder)) and
 (bidder/following::bidder and bidder/preceding::bidder)]
/interval
```

B11(i): `//open_auction(/bidder/..)`^{i}`/interval`

B12(i): `//item(/@id/..)`^{i}`/name`

B13(i): `//keyword(/ancestor::parlist/descendant::keyword)`^{i}

B14(i): `//bidder(/following-sibling::bidder/preceding-sibling::bidder)`^{i}

B15(i): `//keyword(/following::keyword/preceding::keyword)`^{i}

A.3 Comparison Queries

The queries in this section perform data comparisons, some of which use the `id` function, that return all element nodes with an `ID` value that matche the values of the argument to the `id` function.

C1:
```
/site/people/person
[profile/age >= 18 and profile/@income < 10000
```

```
        and address/city != "Dallas"]/name
C2:     /site/open_auctions/open_auction[bidder/increase = current]
        /interval
C3:     /site/people/person
        [profile/@income = /site/open_auctions/open_auction/current]
        /name
C4:     /site/people/person
        [watches/watch/id(@open_auction)/seller/@person = @id]/name
C5:     id("person0")/name
C6:     /site/people/person/watches/watch/id(@open_auction)/interval
C7:     /site/people/person[watches/watch/id(@open_auction)/itemref
        /id(@item)/parent::australian]/name
C8(i):  Y(i)/id(.)/name
          where Y(1) = /site/catgraph/edge[@from = "category0"]/@to
                Y(i) = /site/catgraph/edge[@from = Y(i − 1)]/@to for i ≥ 2
C9(i):  /site/open_auctions/open_auction
        (/seller/id(@person)/watches/watch/id(@open_auction))^i/interval
C10(i): /site/people/person
        (/watches/watch/id(@open_auction)/bidder/personref
        /id(@person))^i/name
```

A.4 Aggregation Queries

Aggregation queries perform arithmetic operations and count and sum up values.

```
D1: /site/open_auctions/open_auction[(count(bidder) mod 2) = 0]
    /interval
D2: count(//text) + count(//bold) + count(//emph) + count(//keyword)
D3: /site/open_auctions/open_auction
    [sum(bidder/increase) > 10 * initial]/interval
D4: /site/open_auctions/open_auction
    [sum(bidder/increase) != (current - initial)]/interval
D5: /site/open_auctions/open_auction
    [bidder and
     (sum(bidder/increase) div count(bidder)) > 2 * initial]/interval
```

A.5 Position Queries

Position queries use positions, see Section 5.3.

```
E1: site/open_auctions/open_auction
    [bidder[1]/number(increase) < bidder[floor((last() + 1) div 2)]
    /number(increase) and bidder[floor((last() + 1) div 2)]
    /number(increase) < bidder[last()]/number(increase)]/interval
```

E2: `/site/regions/europe/item/description/descendant::keyword[last()]`
E3: `//keyword/ancestor::listitem[1]/text/keyword`
E4:
```
site/open_auctions/open_auction/bidder
    [preceding-sibling::bidder[1]/number(increase)
     <= number(increase) and number(increase)
     <= following-sibling::bidder[1]/number(increase)]
```
E5:
```
/site/regions/*/item
    [preceding::item[100] and following::item[100]]/name
```
E6: `/site/regions/*/item[contains(description, name)]/name`
E7:
```
/site/regions/*/item
    [contains(substring-before(description, "eros"), "passion") and
     contains(substring-after(description, "eros"), "dangerous")]
    /name
```
E8:
```
/site/regions/*/item
    [string-length(translate(normalize-space(description)," ",""))
     > 10000]/name
```

A.6 Closure Queries

Closure queries perform the closure from nodes in `$input` and XPath query `following-sibling::bidder[position()=1 and number(increase) <= 10]` according to the function `closure` below, where

Bigger $=_{\text{df}}$
```
following-sibling::bidder[position()=1 and number(increase) > 10]
```

Smaller $=_{\text{df}}$
```
following-sibling::bidder[position()=1 and number(increase) <= 10]
```

and

closure $=_{\text{df}}$
```
function closure($input as node()*, $result as node()*) as node()*
{
 let $current := $input/following-sibling::bidder
                   [position()=1 and number(increase) <= 10]
 let $new := $current except $result
 let $all := ($result,$new)

 return
     if(exists($new))
     then ($new, closure($new,$all))
     else ()
}
```

F1: `//bidder`
`[number(increase) <= 10 and (`*Bigger* `or` *closure*`(.,`*Smaller*`)/`*Bigger*`)]`

F2: `//bidder`
`[number(increase) <= 10 and (`*Bigger* `or` *closure*`(.,`*Smaller*`)/`*Bigger*`)]`

F3: `//listitem[text/keyword or`
closure`(.,parlist/listitem/parlist/listitem)/text/keyword]`
`/text/keyword`

F4: `/site/open_auctions/open_auction[position() <= 5]`
`/`*closure*`(.,seller/id(@person)/watches/watch/id(@open_auction))`
`/interval`

F5: `/site/people/person[position() <= 5]/`*closure*`(.,/watches/watch`
`/id(@open_auction)/bidder/personref/id(@person))/name`

F6: `/site/people/person[1]/idref(@id)/..`

F7: `//category[@id="category0"]/@id`
`/`*closure*`(.,let $i = .  return ($i | //edge[@from = $i]/@to))`
`/id(.)/name`

F8: `//category[@id="category0"]/@id`
`/`*closure*`(.,idref(.)[name() = "from"]/../@to)/id(.)/name`

Appendix B

XPath to λXP Compiler

Contents

We translate XPath 3.0 queries to λXP expressions along its grammar in Ebnf given in Section 5.1 for basic, navigational, postfix, and terminal expressions, and for sequence types whose production rules are given in Section 4.1.

⚠ The following translations were done to the best of our understanding of the informally described XPath 3.0 specification[1].

The translation follows the specification where the XPath 1.0 compatibility mode is set to *false*, and where its type system is based on Xml Schema 1.0. Notice that we can also translate XPath 3.0 expressions where the XPath 1.0 compatibility mode is set to *true*, which then implements different specifications. We use some constant *mode* which is equal to the sort function *sort* if the ordering mode of XQuery is set to *ordered*, while it is the identity function *id*, when the ordering mode is set to *unordered*:

$$mode = \begin{cases} sort & \text{, ordering mode of XQuery is } ordered \\ id & \text{, ordering mode of XQuery is } unordered \end{cases}$$

We assume that XPath expressions are not abbreviated, i.e. we only translate the unabbreviated XPath syntax. Notice that we raise an *error* for expressions that depend on the unsupported namespace axis (see the Xml data model in Section 2.3.1).

[1] https://www.w3.org/TR/xpath-30/

B.1 Basic Expressions

B.1.1 XPath expressions

Nonterminal `XPath` expands to nonterminal `Expr`.

(1) $[\![\texttt{XPath(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{Expr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.1.2 Nonterminals to define inline function expressions

The next four nonterminals in the grammar are used to define function declarations by expressions `InlineFunctionExpr` (64), and thus translated as a whole there.

(2) `ParamList ::= Param ("," Param)*`
(3) `Param ::= "$" EQName TypeDeclaration?`
(4) `FunctionBody ::= EnclosedExpr`
(5) `EnclosedExpr ::= "{" Expr "}"`

B.1.3 Sequence constructions

(6) $[\![\texttt{Expr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{ExprSingle(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

Sequences in XPath are constructed using the comma seperator and they are flattened according to Section 4.3 for which we append the result lists.

$[\![\texttt{Expr(P}_1\texttt{,...,P}_n\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$\mathit{listconcat}([\![\texttt{ExprSingle(P}_1\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}},\ldots,[\![\texttt{ExprSingle(P}_n\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})$

B.1.4 Single expressions

There exist various single expressions.

(7) $[\![\texttt{ExprSingle(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{ForExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{ForExpr} \\ [\![\texttt{LetExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{LetExpr} \\ [\![\texttt{QuantifiedExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{QuantifiedExpr} \\ [\![\texttt{IfExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{IfExpr} \\ [\![\texttt{OrExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{OrExpr} \end{cases}$$

B.1.5 Sequence decompositions

For expressions using multiple variables are first expanded to a set of nested for expressions:

```
ForExpr(for $VarName(y1) in ExprSingle(P1),...,
            $VarName(yn) in ExprSingle(Pn) return ExprSingle(P))
=ForExpr(for $VarName(y1) in ExprSingle(P1) return ...
           for $VarName(yn) in ExprSingle(Pn) return ExprSingle(P))
```

Then single-variable for expressions concatenate sequences by the return expression `P'` in

the order of the items in the binding sequence P:

(8) $[\![$`ForExpr(for $VarName(y) in ExprSingle(P) return ExprSingle(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

listconcat (*map* ($\lambda$$[\![$`VarName(y)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$.$[\![$`ExprSingle(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$) $[\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$)

The following two nonterminals are used to define for expressions `ForExpr`, translated above.

(9) `SimpleForClause ::= "for" SimpleForBinding ("," SimpleForBinding)*`

(10) `SimpleForBinding ::= "$" VarName "in" ExprSingle`

B.1.6 Let expressions

Similar to for expression, also let expressions using multiple variables are first expanded to a set of nested let expressions:

`LetExpr(let $VarName(`y_1`) := ExprSingle(`P_1`),...,`
`$VarName(`y_n`) := ExprSingle(`P_n`) return ExprSingle(P))`
`=LetExpr(let $VarName(`y_1`) := ExprSingle(`P_1`) return ...`
`let $VarName(`y_n`) := ExprSingle(`P_n`) return ExprSingle(P))`

Single-variable let expressions translate to let definitions in λXP:

(11) $[\![$`LetExpr(let $VarName(y):= ExprSingle(P) return ExprSingle(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

let $[\![$`VarName(y)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = [\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in** $[\![$`ExprSingle(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

The following two nonterminals are used to define let expressions `LetExpr`, translated above.

(12) `SimpleLetClause ::= "let" SinleLetBinding ("," SimpleLetBinding)*`

(13) `SimpleLetBinding ::= "$" VarName ":=" ExprSingle`

B.1.7 Quantified expressions

Quantified expressions that we mentioned briefly in Section 5.2.2 can bind several variables in general. Expression `P` below that follows the `satisfies` keyword may be tested for any such variable binding from variables $\mathtt{y}_i$ to items of sequences $\mathtt{P}_i$. The translation to λXP expressions is therefore a little tedious. It obtains n lists of elements from expressions $\mathtt{P}_i$, which must be decomposed, such that all needed variable bindings are obtained. This requires to re-iterate through certain lists in general, which is done by the functions in the following translations. When function f_i will be called, variables $\mathtt{y}_1$ until $\mathtt{y}_{i-1}$ have been bound by previous calls to functions f_j for $j < i$ via the let definition for those variables. Function f_i then iterates through the list by $\mathtt{P}_i$ while binding variable $\mathtt{y}_i$ and which evaluates f_{i+1} if $i < n$, or if $i = n$ it evaluates `P`, since all appearing variables are bound. When one of these evaluations return *true*, the function returns *true* and stops the iteration.

(14) $[\![$`QuantifiedExpr(some`
`$VarName(`y_1`) in ExprSingle(`P_1`), ...,`
`$VarName(`y_n`) in ExprSingle(`P_n`) satisfies ExprSingle(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $f_n = \lambda z_n$.**match** z_n **with** $x :: y$ **then**
let $[\![$`VarName(`y_n`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $[\![$`fn:boolean`$]\!]([\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ **then** *true* **else** $f_n(y)$
else *false*
in let $f_{n-1} = \lambda(z_{n-1}, z_n)$.**match** z_{n-1} **with** $x :: y$ **then**
let $[\![$`VarName(`y_{n-1}`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $f_n(z_n)$ **then** *true* **else** $f_{n-1}(y, z_n)$
else *false*
in ...
in let $f_1 = \lambda(z_1, \ldots, z_n)$.**match** z_1 **with** $x :: y$ **then**
let $[\![$`VarName(`y_1`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $f_2(z_2, \ldots, z_n)$ **then** *true* **else** $f_1(y, z_2, \ldots, z_n)$
else *false*
in $f_1([\![$`ExprSingle(`P_1`),...,ExprSingle(`P_n`))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

In contrast to test for the existence of a variable binding for which expressions `P` becomes *true* as above, the quantification with keyword `every` tests whether `P` becomes *true* for all variables bindings:

$[\![$`QuantifiedExpr(every`
`$VarName(`y_1`) in ExprSingle(`P_1`), ...,`
`$VarName(`y_n`) in ExprSingle(`P_n`) satisfies ExprSingle(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $f_n = \lambda z_n$.**match** z_n **with** $x :: y$ **then**
let $[\![$`VarName(`y_n`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $[\![$`fn:boolean`$]\!]([\![$`ExprSingle(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ **then** $f_n(y)$ **else** *false*
else *true*
in let $f_{n-1} = \lambda(z_{n-1}, z_n)$.**match** z_{n-1} **with** $x :: y$ **then**
let $[\![$`VarName(`y_{n-1}`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $f_n(z_n)$ **then** $f_{n-1}(y, z_n)$ **else** *false*
else *true*
in ...
in let $f_1 = \lambda(z_1, \ldots, z_n)$.**match** z_1 **with** $x :: y$ **then**
let $[\![$`VarName(`y_1`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = x$ **in**
if $f_2(z_2, \ldots, z_n)$ **then** $f_1(y, z_2, \ldots, z_n)$ **else** *false*
else *true*
in $f_1([\![$`ExprSingle(`P_1`),...,ExprSingle(`P_n`))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.1.8 First-Order expressions

Conditionals, disjunctions, and conjunctions of XPath are translated to conditionals, disjunctions, and conjunctions in λXP, respectively.

(15) $[\![$`IfExpr(if (Expr(P)) then ExprSingle(P') else ExprSingle(P''))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
if $[\![$`fn:boolean`$]\!]([\![$`Expr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$
then $[\![$`ExprSingle(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ **else** $[\![$`ExprSingle(P'')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

(16) $[\![$`OrExpr(AndExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`AndExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`OrExpr(AndExpr(`$\mathtt{P}_1$`) or ... or AndExpr(`$\mathtt{P}_n$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$[\![$`fn:boolean`$]\!]([\![$`AndExpr(`$\mathtt{P}_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}}) \vee \ldots \vee [\![$`fn:boolean`$]\!]([\![$`AndExpr(`$\mathtt{P}_n$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$

(17) $[\![$`AndExpr(ComparisonExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`ComparisonExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`AndExpr(ComparisonExpr(`$\mathtt{P}_1$`) and ... and ComparisonExpr(`$\mathtt{P}_n$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$[\![$`fn:boolean`$]\!]([\![$`ComparisonExpr(`$\mathtt{P}_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$
$\wedge \ldots \wedge [\![$`fn:boolean`$]\!]([\![$`ComparisonExpr(`$\mathtt{P}_n$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$

B.1.9 Comparisons

Comparison expressions may expand to string concatenation expressions.
(18) $[\![$`ComparisonExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`StringConcatExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

Otherwise we translate comparison expression to λXP expressions as follows. It spells out the details of the XPath 3.0 specifications, whose essence we described in Section 5.6. The translation assumes a function *lct* that computes the least common type of two types in XPath. It also assumes a function *validComb* that tests whether the transformed operands are of valid type with respect to the operator according to the specification[2].

[2] http://www.w3.org/TR/xpath-30/#mapping

$$
\begin{array}{l}
[\![\texttt{ComparisonExpr(}\\
\texttt{StringConcatExpr(P) ValueComp(vc) StringConcatExpr(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\mathrm{df}}\\
\quad \textbf{let } \mathit{right} = [\![\texttt{fn:data}]\!]([\![\texttt{StringConcatExpr(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}) \textbf{ in}\\
\quad \textbf{match } [\![\texttt{fn:data}]\!]([\![\texttt{StringConcatExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}) \textbf{ with } x_1 :: y_1 \textbf{ then}\\
\qquad \textbf{if } \mathit{right} = \mathit{nil} \textbf{ then } \mathit{nil} \textbf{ else}\\
\qquad\quad \textbf{if } y_1 = \mathit{nil} \textbf{ then}\\
\qquad\qquad \textbf{match } \mathit{right} \textbf{ with } x_2 :: y_2 \textbf{ then}\\
\qquad\qquad\quad \textbf{if } y_2 = \mathit{nil} \textbf{ then}\\
\qquad\qquad\qquad \textbf{let } x_1' = \textbf{if } \mathit{type}_{\mathrm{TYPE}(x_1),\textbf{xs:untypedAtomic}}(x_1)\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathit{cast}_{\textbf{xs:untypedAtomic},\textbf{string}}(x_1) \textbf{ else } x_1 \textbf{ in}\\
\qquad\qquad\qquad \textbf{let } x_2' = \textbf{if } \mathit{type}_{\mathrm{TYPE}(x_2),\textbf{xs:untypedAtomic}}(x_2)\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathit{cast}_{\textbf{xs:untypedAtomic},\textbf{string}}(x_2) \textbf{ else } x_2 \textbf{ in}\\
\qquad\qquad\qquad \textbf{let } x_1'' = \mathit{cast}_{\mathrm{TYPE}(x_1'),\mathit{lct}(\mathrm{TYPE}(x_1'),\mathrm{TYPE}(x_2'))}(x_1') \textbf{ in}\\
\qquad\qquad\qquad \textbf{let } x_2'' = \mathit{cast}_{\mathrm{TYPE}(x_2'),\mathit{lct}(\mathrm{TYPE}(x_1'),\mathrm{TYPE}(x_2'))}(x_2') \textbf{ in}\\
\qquad\qquad\qquad \textbf{let } \mathit{op} = [\![\texttt{ValueComp(vc)}]\!]_{\bar{z},\hat{z},\tilde{z}} \textbf{ in}\\
\qquad\qquad\qquad \textbf{if } \mathit{validComb}(\mathrm{TYPE}(x_1''), \mathrm{TYPE}(x_2''), \mathit{op})\\
\qquad\qquad\qquad\quad \textbf{then } x_1'' \ \mathit{op}\ x_2'' \textbf{ else raise } \mathit{error}(\mathrm{err{:}XPTY0004})\\
\qquad\qquad\quad \textbf{else raise } \mathit{error}(\mathrm{err{:}XPTY0004})\\
\qquad\qquad \textbf{else } \mathit{nil}\\
\qquad\quad \textbf{else raise } \mathit{error}(\mathrm{err{:}XPTY0004})\\
\quad \textbf{else } \mathit{nil}
\end{array}
$$

$[\![$`ComparisonExpr(`
`StringConcatExpr(P) GeneralComp(gc) StringConcatExpr(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $x_1 = [\![$`fn:data`$]\!]([\![$`StringConcatExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$,
$op = [\![$`GeneralComp(gc)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$,
$x_2 = [\![$`fn:data`$]\!]([\![$`StringConcatExpr(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ **in**
$\exists x.[x \in list2set(x_1) \wedge \exists y.[y \in list2set(x_2) \wedge$
let $x' =$ **if** $type_{\text{TYPE}(x),\mathbf{xs:untypedAtomic}}(x) \wedge type_{\text{TYPE}(y),\mathbf{xs:untypedAtomic}}(y)$
then $cast_{\mathbf{xs:untypedAtomic},\mathbf{string}}(x)$ **else** x **in**
let $y' =$ **if** $type_{\text{TYPE}(x),\mathbf{xs:untypedAtomic}}(x) \wedge type_{\text{TYPE}(y),\mathbf{xs:untypedAtomic}}(y)$
then $cast_{\mathbf{xs:untypedAtomic},\mathbf{string}}(y)$ **else** y **in**
if $type_{\text{TYPE}(x'),\mathbf{xs:untypedAtomic}}(x') \vee type_{\text{TYPE}(y'),\mathbf{xs:untypedAtomic}}(y') \vee$ **then**
let $left =$ **if** $type_{\text{TYPE}(x'),\mathbf{xs:untypedAtomic}}(x')$ **then** x' **else** y' **in**
let $right =$ **if** $type_{\text{TYPE}(x'),\mathbf{xs:untypedAtomic}}(x')$ **then** y' **else** x' **in**
let $left' =$
if $type_{\text{TYPE}(left),\mathbf{num}}(left)$ **then** $cast_{\text{TYPE}(left),\mathbf{xs:double}}(left)$
else if $type_{\text{TYPE}(left),\mathbf{xs:dayTimeDuration}}(left)$ **then**
$cast_{\text{TYPE}(left),\mathbf{xs:dayTimeDuration}}(left)$
else if $type_{\text{TYPE}(left),\mathbf{xs:yearMonthDuration}}(left)$ **then**
$cast_{\text{TYPE}(left),\mathbf{xs:yearMonthDuration}}(left)$
else $cast_{\text{TYPE}(left),\text{TYPE}(right)}(left)$
in if $validComb(\text{TYPE}(left'), \text{TYPE}(right), op)$
then $left'$ op $right$ **else raise** $error$
else if $validComb(\text{TYPE}(x'), \text{TYPE}(y'), op)$ **then** x' op y' **else raise** $error$

$[\![$`ComparisonExpr(`
`StringConcatExpr(P) NodeComp(nc) StringConcatExpr(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $left = [\![$`StringConcatExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$,
$right = [\![$`StringConcatExpr(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
if $left = nil \vee right = nil$ **then** nil **else**
match $left$ **with** $x_1 :: y_1$ **then**
if $y_1 = nil \wedge type_{\text{TYPE}(x_1),\mathbf{node},}(x_1)$ **then**
match $right$ **with** $x_2 :: y_2$ **then**
if $y_2 = nil \wedge type_{\text{TYPE}(x_2),\mathbf{node}}(x_2)$ **then**
x_1 $[\![$`NodeComp(nc)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ x_2
else raise $error$(//err:XPTY0004)
else nil
else raise $error$(//err:XPTY0004)
else nil

B.1.10 Concatenation of strings

(19) $[\![$`StringConcatExpr(RangeExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`RangeExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$
String concatenation expressions are translated by XPath 3.0 function `fn:concat`,

i.e. $a||$b = fn:concat($a,$b).

$[\![\texttt{StringConcatExpr(RangeExpr(P}_1\texttt{) || ...|| RangeExpr(P}_n\texttt{))}]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$[\![\texttt{fn:concat}]\!]([\![\texttt{RangeExpr(P}_1\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}}, \ldots, [\![\texttt{RangeExpr(P}_n\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}})$

B.1.11 Range expressions

We would like to refer to Section 5.3 for an example of a range expression, who evaluate to sequence of consecutive integers.

(20) $[\![\texttt{RangeExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{AdditiveExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![\texttt{RangeExpr(AdditiveExpr(P) to AdditiveExpr(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
match $[\![\texttt{AdditiveExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **with** $x_1 :: y_1$ **then**
match $[\![\texttt{AdditiveExpr(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **with** $x_2 :: y_2$ **then**
let $left = [\![\texttt{fn:number}]\!](x_1), right = [\![\texttt{fn:number}]\!](x_2)$ **in**
if $left > right$ **then** nil **else** $left :: \ldots :: right :: nil$
else nil
else nil

B.1.12 Arithmetic Expressions

Operands of arithmetic expressions undergo atomization, see Section 5.5.1. The translation for additive and multiplicate expressions tests whether all operands and operators are pairwise valid combinations ($validComb(z_1, \ldots, z_{n+1}, \texttt{op}_1, \ldots, \texttt{op}_n)$). For additive expressions `AdditiveExpr` operators are $\texttt{op}_i \in \{\texttt{+}, \texttt{-}\}$ for all $1 \leq i \leq n$:

(21) $[\![\texttt{AdditiveExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{MultiplicativeExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`AdditiveExpr(`
`MultiplicativeExpr(`$\mathtt{P}_1$`)` $\mathtt{op}_1$ `MultiplicativeExpr(`$\mathtt{P}_2$`)` ...
`MultiplicativeExpr(`$\mathtt{P}_n$`)` $\mathtt{op}_n$ `MultiplicativeExpr(`$\mathtt{P}_{n+1}$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $x_1 = [\![$`fn:data`$]\!]([\![$`MultiplicativeExpr(`$\mathtt{P}_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}}), \ldots,$
$x_{n+1} = [\![$`fn:data`$]\!]([\![$`MultiplicativeExpr(`$\mathtt{P}_{n+1}$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ **in**
if $x_1 = \mathit{nil} \vee \ldots \vee x_{n+1} = \mathit{nil}$ **then** nil **else**
let $z_1 =$ **match** x_1 **with** $x :: y$ **then**
if $\mathit{type}_{\text{TYPE}(x),\mathbf{xs:untypedAtomic}}(x)$ **then**
try $\mathit{cast}_{\text{TYPE}(x),\mathbf{xs:double}}(x)$ **catch**(e) **raise** error(err:FORG0001)
else x
else raise error(err:XPTY0004) **in** ...
let $z_{n+1} =$ **match** x_{n+1} **with** $x :: y$ **then**
if $\mathit{type}_{\text{TYPE}(x),\mathbf{xs:untypedAtomic}}(x)$ **then**
try $\mathit{cast}_{\text{TYPE}(x),\mathbf{xs:double}}(x)$ **catch**(e) **raise** error(err:FORG0001)
else x
else raise error(err:XPTY0004) **in** ...
if $\mathit{validComb}(z_1, \ldots, z_{n+1}, \mathtt{op}_1, \ldots, \mathtt{op}_n)$ **then**
try z_1 $\mathtt{op}_1$ z_2 $\mathtt{op}_2$ ... z_n $\mathtt{op}_n$ z_{n+1} **catch**(e) **raise** $\mathit{error}(e)$
else raise error(err:XPTY0004)

We assume that the operators belong to the built-in contansts $\mathcal{C}onst$ in λXP. Similar to additive expressions we translate multiplicative expressions `MultiplicativeExpr`, but where $\mathtt{op}_i \in \{$`*`, `div`, `idiv`, `mod`$\}$ for all $1 \leq i \leq n$:

(22) $[\![$`MultiplicativeExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`UnionExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$
$[\![$`MultiplicativeExpr(`
`UnionExpr(`$\mathtt{P}_1$`)` $\mathtt{op}_1$ `UnionExpr(`$\mathtt{P}_2$`)` ...
`UnionExpr(`$\mathtt{P}_n$`)` $\mathtt{op}_n$ `UnionExpr(`$\mathtt{P}_{n+1}$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
same translation as for AdditiveExpr((21)), but where
$x_i = [\![$`fn:data`$]\!]([\![$`UnionExpr(`$\mathtt{P}_i$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ for all $1 \leq i \leq n+1$

B.1.13 Ordered Sets

The following two nonterminals `UnionExpr` and `IntersectExceptExpr` produce operators `union`, `intersect`, and `except`, which operate on sequences of nodes only, see Section 5.7.1. Notice that according to the specification, if one of the items in the sequences is not a node, a type error is raised, which is treated in shortcut *nodelist2set*. Furthermore the result list may be sorted in document order according to constant *mode*.

(23) $[\![$`UnionExpr(IntersectExceptExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`IntersectExceptExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

For the following union operator `union`, in XPath also `|` may be written, which has the same semantics.

$[\![$`UnionExpr(IntersectExceptExpr(`$\mathtt{P}_1$`) union ... union IntersectExceptExpr(`$\mathtt{P}_n$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$mode(set2list(nodelist2set([\![$`IntersectExceptExpr(`$\mathtt{P}_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}}) \cup \ldots \cup nodelist2set([\![$`IntersectExceptExpr(`$\mathtt{P}_n$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})))$

(24) $[\![$`IntersectExceptExpr(InstanceofExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`InstanceofExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

For expressions `IntersectExceptExpr` we have $\mathtt{op}_i \in \{\mathtt{intersect}, \mathtt{except}\}$ for all $1 \leq i \leq n$:

$[\![$`IntersectExceptExpr(`
`InstanceofExpr(`$\mathtt{P}_1$`)` $\mathtt{op}_1$ `InstanceofExpr(`$\mathtt{P}_2$`) ...`
`InstanceofExpr(`$\mathtt{P}_n$`)` $\mathtt{op}_n$ `InstanceofExpr(`$\mathtt{P}_{n+1}$`))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =^{n\geq 2}_{\text{df}}$
$mode(set2list($
$nodelist2set([\![$`InstanceofExpr(`$\mathtt{P}_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ $\mathtt{op}_1$
$nodelist2set([\![$`InstanceofExpr(`$\mathtt{P}_2$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ $\mathtt{op}_2$
$\ldots$ $\mathtt{op}_n$ $nodelist2set([\![$`InstanceofExpr(`$\mathtt{P}_{n+1}$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})))$

B.1.14 Type checks and casts

We refer to Section 4.4 for some examples.

The instance-of expression `P instance of P'` tests whether the type of the result by `P` matches the type by `P'`, for which it is reduced to sequence type matching in Section B.4.

(25) $[\![$`InstanceofExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`TreatExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`InstanceofExpr(TreatExpr(P) instance of SequenceType(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`SequenceType(P')`$]\!]^{[\![\mathtt{TreatExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}}_{\bar{z},\hat{z},\tilde{z}}$

Treat expressions in XPath varify whether an expressions is of a certain type. If the value of the expression matches the type, then it is returned, otherwise an error is raised.

(26) $[\![$`TreatExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`CastableExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`TreatExpr(CastableExpr(P) treat as SequenceType(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
if $[\![$`InstanceofExpr(CastableExpr(P) instance of SequenceType(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$
then $[\![$`CastableExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ **else raise** *error*(err:XPDY0050)

As mentioned in Section 5.8 the expressions that varify whether some value is castable into a certain type, requires a facility to try and catch errors.

(27) $[\![$`CastableExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`CastExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`CastableExpr(CastExpr(P) castable as SingleType(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
try
let $z = [\![$`CastExpr(P cast as SingleType(P'))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in** *true*
catch(e) *false*

Cast expressions in XPath are modeled with constant cast functions $cast_{S,T} \in$

Const in λXP that are defined for every combination of source types S and target types T according to the specification[3].

(28) $[\![\texttt{CastExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{UnaryExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![\texttt{CastExpr(UnaryExpr(P) cast as SimpleTypeName(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $z = [\![\texttt{fn:data}]\!]([\![\texttt{UnaryExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}})$ **in**
match z **with** $x :: y$ **then**
if $y = nil$ **then** $cast_{\text{TYPE}(x),\text{TYPE}([\![\texttt{SimpleTypeName}]\!]_{\bar{z},\hat{z},\tilde{z}})}(x)$
else raise *error*(err:XPTY0004)
else raise *error*

$[\![\texttt{CastExpr(UnaryExpr(P) cast as SimpleTypeName(P')?)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $z = [\![\texttt{fn:data}]\!]([\![\texttt{UnaryExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}})$ **in**
match z **with** $x :: y$ **then**
if $y = nil$ **then** $cast_{\text{TYPE}(x),\text{TYPE}([\![\texttt{SimpleTypeName}]\!]_{\bar{z},\hat{z},\tilde{z}})}(x)$
else raise *error*(err:XPTY0004)
else *nil*

B.1.15 Unary Expressions

(29) $[\![\texttt{UnaryExpr(ValueExpr(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{ValueExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

Any sequence of unary operators `+` and `-` is reduced to one unary operator in the obvious way. The translation is similar to the translation of arithmetic expressions as above. $\texttt{op} \in \{\texttt{+}, \texttt{-}\}$.

$[\![\texttt{UnaryExpr(op ValueExpr(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
match $[\![\texttt{fn:data}]\!]([\![\texttt{ValueExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}})$ **with** $x :: y$ **then**
if $y = nil$ **then**
let $x' =$ **if** $type_{\text{TYPE}(x),\textbf{xs:untypedAtomic}}(x)$
then try $cast_{\text{TYPE}(x),\textbf{xs:double}}(x)$ **catch**(e) **raise** *error*(err:FORG0001)
else x
in
if *validComb*$(x', \texttt{op})$ **then**
$\texttt{op}\ [\![\texttt{fn:number}]\!](x')$ **else raise** *error*(err:XPTY0004)
else raise *error*(err:XPTY0004)
else *nil*

B.1.16 Value expressions

Value expressions are simple map expressions.

(30) $[\![\texttt{ValueExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{SimpleMapExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

[3] http://www.w3.org/TR/xpath-functions-30/#casting

B.1.17 Comparison operators

The translation of general comparison, value comparison, and node comparison operators is straight forward.

(31) $[\![\texttt{GeneralComp(=)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} =$
$[\![\texttt{GeneralComp(!=)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \neq$
$[\![\texttt{GeneralComp(<)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} <$
$[\![\texttt{GeneralComp(<=)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \leq$
$[\![\texttt{GeneralComp(>)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} >$
$[\![\texttt{GeneralComp(>=)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \geq$

(32) $[\![\texttt{ValueComp(eq)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} =$
$[\![\texttt{ValueComp(ne)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \neq$
$[\![\texttt{ValueComp(lt)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} <$
$[\![\texttt{ValueComp(le)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \leq$
$[\![\texttt{ValueComp(gt)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} >$
$[\![\texttt{ValueComp(ge)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \geq$

(33) $[\![\texttt{NodeComp(is)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} =$
$[\![\texttt{NodeComp(<<)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} <$
$[\![\texttt{NodeComp(>>)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} >$

B.1.18 Simple Map expressions

(34) $[\![\texttt{SimpleMapExpr(PathExpr(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{PathExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

The translation of the simple map operator (!) from Section 3.3.6 makes the treatment of context nodes ci, context positions cp, and context size cs explicite using nested data structures, as we discussed in Section 7.3.2.

$[\![\texttt{SimpleMapExpr(PathExpr(P) ! PathExpr(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $apply = \lambda z.$**match** z **with** $x :: y$ **then match** x **with** (ci, cp, cs) **in**
if $type_{\text{TYPE}(ci),\textbf{node}}(ci)$ **then**
$\textbf{at}(ci, (\lambda(z^i, z^p, z^s).[\![\texttt{PathExpr(P')}]\!]_{z^i,z^p,z^s})(ci, cp, cs)) :: apply(y)$
else
$(\lambda(z^i, z^p, z^s).[\![\texttt{PathExpr(P')}]\!]_{z^i,z^p,z^s})(ci, cp, cs) :: apply(y)$
else nil **in**
$listconcat\, apply(addPositionAndSize([\![\texttt{PathExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

However for reverse axis, the assignment of context positions is in reverse document order:

$[\![$`ReverseAxis(P)::NodeTest(P') ! PathExpr(P'')`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let *apply* $= \lambda z.$**match** z **with** $x :: y$ **then**
match x **with** (ci, cp, cs) **in**
at$(ci, (\lambda(z^i, z^p, z^s).[\![$`PathExpr(P'')`$]\!]_{z^i,z^p,z^s})(ci, cp, cs)) :: apply(y)$
else *nil*
in *listconcat apply*(
addReversePositionAndSize($[\![$`ReverseAxis(P)::NodeTest(P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$))

B.2 Navigational Expressions

B.2.1 Absolute path expressions

A "/" at the beginning of a path expression is an abbreviation for the root node of the tree to which the context node `self::node()` belongs.

(35) $[\![$`PathExpr(/)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`PathExpr((fn:root(self::node()) treat as document-node()))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`PathExpr(/RelativePathExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`PathExpr((fn:root(self::node()) treat as document-node())`
`/RelativePathExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![$`PathExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`RelativePathExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.2.2 Relative path expressions

(36) $[\![$`RelativePathExpr(StepExpr(P))`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`StepExpr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

Each occurence of `//` in relative path expressions `RelativePathExpr` is an abbreviation for `/descendant-or-self::node()/`.

$[\![$`RelativePathExpr(P//P')`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`RelativePathExpr(P/descendant-or-self::node()/P')`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

The path operator `/` that we describe in Section 3.3.6 can be expressed using the simple map operator `!`:

$[\![$`RelativePathExpr(P/P')`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![$`RelativePathExpr(`
```
let $R:= P!P' return
  if (every $r in $R satisfies $r instance of node()) then
    ($R union ())
  else if (every $r in $R satisfies not($r instance of node()))
    then $R
  else error()
```
$]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.2.3 Steps

Steps in XPath 3.0 are expressions that produce a sequence of items that may be filtered. A step is either an axis step or a postfix expression.

(37) $[\![\texttt{StepExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$$\begin{cases} [\![\texttt{PostfixExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{PostfixExpr} \\ [\![\texttt{AxisStep(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{AxisStep} \end{cases}$$

B.2.4 Axis steps and Filters

An axis step, described in Section 3.3.1, is either a forward or a backward step.

(38) $[\![\texttt{AxisStep(ForwardStep(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{ForwardStep(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

$[\![\texttt{AxisStep(ReverseStep(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{ReverseStep(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

An axis steps with predicates or postfix expressions `PostfixExpr` (48) are translated as follows. The expression `P` below is some expression $\bar{\texttt{P}}$ of either a forward step `AxisStep(ForwardStep(`$\bar{\texttt{P}}$`))`, a backward step `AxisStep(ReverseStep(`$\bar{\texttt{P}}$`))`, or a primary expression `PostfixExpr(PrimaryExpr(`$\bar{\texttt{P}}$`))`. More details on filters can be found in Section 3.3.5. Similarly to the translation of the simple map operator, it must take care of context nodes and positions explicitely.

$[\![\texttt{P[P}']\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

let $eval = \lambda z.$**match** z **with** $x :: y$ **then**
  match x **with** (ci, cp, cs) **in**
    let *filter* = **if** $type_{\text{TYPE}(ci),\textbf{node}}(ci)$ **then**
        at$(ci, (\lambda(z^i, z^p, z^s).[\![\texttt{P}']\!]_{z^i,z^p,z^s})(ci, cp, cs))$
      else
        $(\lambda(z^i, z^p, z^s).[\![\texttt{P}']\!]_{z^i,z^p,z^s})(ci, cp, cs)$
    in
      if $type_{\text{TYPE}(\mathit{filter}),\textbf{num}}(\mathit{filter}) \wedge cast_{\text{TYPE}(\mathit{filter}),\textbf{int}}(\mathit{filter}) = cp$ **then**
        $ci :: eval(y)$
      else if $[\![\texttt{fn:boolean}]\!](\mathit{filter})$ **then**
        $ci :: eval(y)$
      else $eval(y)$
  else *nil*
in $eval(addPositionAndSize([\![\texttt{P}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

Notice that context positions are assigned in reverse document order for testing predicated on nodes selected by reverse axes:

$[\![\texttt{ReverseAxis(P)::NodeTest(P')[Expr(P'')]}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $eval = \lambda z.$**match** z **with** $x :: y$ **then**
match x **with** (ci, cp, cs) **in**
let $filter =$ **at**$(ci, (\lambda(z^i, z^p, z^s).[\![\texttt{Expr(P'')}]\!]_{z^i,z^p,z^s})(ci, cp, cs))$ **in**
if $type_{\text{TYPE}(filter),\textbf{num}}(filter) \wedge cast_{\text{TYPE}(filter),\textbf{int}}(filter) = cp$ **then**
$ci :: eval(y)$
else if $[\![\texttt{fn:boolean}]\!](filter)$ **then**
$ci :: eval(y)$
else $eval(y)$
else nil
in $eval(addReversePositionAndSize($
$[\![\texttt{ReverseAxis(P)::NodeTest(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

B.2.5 Axis Steps

The following translations concern axis steps, as discussed in Section 3.3.1. Node label and type tests are reduced to sequence type matching as in Section B.4.

(39) $[\![\texttt{ForwardStep(ForwardAxis(P) NodeTest(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$mode(set2list(\{x \mid x \in [\![\texttt{ForwardAxis(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} \wedge [\![\texttt{NodeTest(P')}]\!]^x_{\bar{z},\hat{z},\tilde{z}}\}))$

Attribute nodes of XML data trees are considered to be children of their parents, in contrast to what the XML data model specifies. Thus the following translation has to take this into account.

(40) $[\![\texttt{ForwardAxis(child::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ch(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(descendant::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ch^+(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(attribute::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ch(is_x) \wedge type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(self::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid is_x\}$
$[\![\texttt{ForwardAxis(descendant-or-self::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ch^*(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(following-sibling::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ns^+(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(following::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid par^*(ns^+(ch^*(is_x))) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ForwardAxis(namespace::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
raise *error*

We do not consider abbreviated XPATH queries. "`@`" is an abbreviation for "`attribute::`".

(41) `AbbrevForwardStep ::= "@"?  NodeTest`

The nonterminal for abbreviated forward steps `AbbrevForwardStep` is not considered.

(42) $[\![\texttt{ReverseStep(ReverseAxis(P) NodeTest(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$mode(set2list(\{x \mid x \in [\![\texttt{ReverseAxis(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} \wedge [\![\texttt{NodeTest(P')}]\!]^{x}_{\bar{z},\hat{z},\tilde{z}}\}))$

(43) $[\![\texttt{ReverseAxis(parent::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid par(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ReverseAxis(ancestor::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid par^{+}(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ReverseAxis(preceding-sibling::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid ps^{+}(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ReverseAxis(preceding::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid par^{*}(ps^{+}(ch^{*}(is_x))) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$
$[\![\texttt{ReverseAxis(ancestor-or-self::)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\{x \mid par^{*}(is_x) \wedge \neg type_{\text{TYPE}(x),\textbf{attribute}}(x)\}$

We do not consider abbreviated XPath queries. "`..`" is an abbreviation for "`parent::node()`".
(44) `AbbrevReverseStep ::= ".."`

B.2.6 Node Tests, Name Tests, and Wildcards

Translation for node tests `NodeTest`, name tests `NameTest`, and wildcards `Wildcard` are parametrized by the node v, for which the test is to be performed. It reduces to sequence type matching in Section B.4.

(45) $[\![\texttt{NodeTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$$\begin{cases} [\![\texttt{KindTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{KindTest} \\ [\![\texttt{NameTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{NameTest} \end{cases}$$

The following translation for name tests `NameTest` uses shortcut *testEQName*.

(46) $[\![\texttt{NameTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$$\begin{cases} \textbf{let } z = [\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} \textbf{ in} \\ \quad testEQName(v, z) & \texttt{P} \in \texttt{EQName} \\ [\![\texttt{Wildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{Wildcard} \end{cases}$$

(47) $[\![\texttt{Wildcard(*)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$labtype(v) = el \vee labtype(v) = att$
$[\![\texttt{Wildcard(NCName(P):*)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{fn:prefix-from-QName}]\!](lab_{name}(v)) = [\![\texttt{NCName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$
$[\![\texttt{Wildcard(*:NCName(P))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{fn:local-name-from-QName}]\!](lab_{name}(v)) = [\![\texttt{NCName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$
$[\![\texttt{Wildcard(BracedURILiteral(P)*)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{fn:prefix-from-QName}]\!](lab_{name}(v)) = prefix([\![\texttt{BracedURILiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}})$

B.3 Postfix expressions

B.3.1 Postfix expressions

(48) $[\![\texttt{PostfixExpr(PrimaryExpr(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{PrimaryExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

Expressions `PostfixExpr(PrimaryExpr(P)[Expr(P')])` are translated as presented before for filters, see `AxisStep` (38).

Argument expressions `PostfixExpr(PrimaryExpr(P)(Argument(`P_1`),...,Argument(`P_n`)))` are translated as function calls `FunctionCall` (59).

B.3.2 Argument Lists and Predicates

Argument lists are used to define function calls, see `FunctionCall` (59) and thus not translated.

(49) `ArgumentList ::= "(" (Argument ("," Argument)*)?  ")"`

Predicates are translated as filters, that may appear with expressions `AxisStep` and `PostfixExpr`, and thus also not translated.

(50) `PredicateList ::= Predicate*`

(51) `Predicate ::= "[" Expr "]"`

B.3.3 Primary expressions

There are the following primary expressions in XPath.

(52) $[\![\texttt{PrimaryExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{Literal(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{Literal} \\ [\![\texttt{VarRef(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{VarRef} \\ [\![\texttt{ParenthesizedExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{ParenthesizedExpr} \\ [\![\texttt{ContextItemExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{ContextItemExpr} \\ [\![\texttt{FunctionCall(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{FunctionCall} \\ [\![\texttt{FunctionItemExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{FunctionItemExpr} \end{cases}$$

B.3.4 Literals

Literals translate to constants in λXP, see Section B.5.

(53) $[\![\texttt{Literal(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{NumericLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{NumericLiteral} \\ [\![\texttt{StringLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{StringLiteral} \end{cases}$$

(54) $[\![\texttt{NumericLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{IntegerLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{IntegerLiteral} \\ [\![\texttt{DecimalLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{DecimalLiteral} \\ [\![\texttt{DoubleLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{DoubleLiteral} \end{cases}$$

B.3.5 Variables

Variables translate to corresponding variables in λXP.

(55) $[\![\texttt{\$VarRef(VarName(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{VarName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(56) $[\![\texttt{VarName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.3.6 Parenthesized expressions and the empty list

Expressions with surrounding parenthesis translate to the expressions without the parenthesis.

(57) $[\![\texttt{(Expr(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{Expr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

The expression `()` translates to the empty list in λXP.

$[\![\texttt{()}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
nil

B.3.7 Context Item, Position, and Size

Whenever an XPath query makes reference to the context item, position, or size in filters or path expressions, then we translate as explained in Section 7.3.2. It evaluates these expressions using some function whose argument will be a triple of context item, position, and size that arrive during runtime. These arguments will substitute the function's parameters $\bar{z}$, $\hat{z}$, and $\tilde{z}$, to which we translate the context item expression, function `fn:position`, and function `fn:last()`, respectively.

(58) $[\![\texttt{ContextItemExpr(.)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \bar{z}$
$[\![\texttt{fn:position()}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \hat{z}$
$[\![\texttt{fn:last()}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \tilde{z}$

B.3.8 Function Calls

Function calls rely on rules for function conversion and coercion, as mentioned in Section 5.2.3. We use a function $\textit{convert-arg}^{ST}(x)$ that converts arguments according to the function conversion rules. It converts arguments x with respect to a target sequence type ST. When ST is a `TypedFunctionTest` (92), then function coercion is applied. Otherwise if ST is a **generalized atomic type** AT, for example, the function converts x as follows:

$\textit{convert-arg}^{AT} =_{\text{df}}$

$\lambda(x).$**let** $x' = [\![\texttt{fn:data}]\!](x)$ **in**
letrec $fun = \lambda z.$
match z **with** $y :: y'$ **then**
if $type_{\text{TYPE}(y),\textbf{xs:untypedAtomic}}(y)$ **then**
if $type_{\text{TYPE}(AT),\textbf{num}}(y)$ **then**
$cast_{\text{TYPE}(y),\textbf{xs:double}}(y) :: fun(y')$
else if $type_{\text{TYPE}(y),\textbf{xs:QName}}(y) \vee type_{\text{TYPE}(y),\textbf{xs:Notation}}(y)$ **then**
raise *error*
else
$cast_{\text{TYPE}(y),\text{TYPE}(AT)}(y) :: fun(y')$
else
$cast_{\text{TYPE}(y),\text{TYPE}(AT)}(y) :: fun(y')$
else *nil*
in $fun(x')$

The following translation for function call expressions `FunctionCall` maps to function applications in λXP in the absence of place holders `?`, and otherwise returns a new function that is obtained via currying. The result of the function call is the value of a function f that matches $[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ when applied to the converted arguments, of which none is a place holder `?`. Otherwise the result is a new function which has as many arguments as place holders:

(59) $[\![\texttt{FunctionCall(EQName(P)(Argument(P}_1\texttt{),...,Argument(P}_n\texttt{)))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\lambda(z_{k_1},\ldots,z_{k_{n'}}).f(m_1,\ldots,m_n)$
where $f : ST_1 \times \ldots \times ST_n \to ST$ is function matching $[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$,
$(m_i) = \begin{cases} \mathit{convert\text{-}arg}^{ST_i}([\![\texttt{ExprSingle(P}_i\texttt{)}]\!]_{\bar{z},\hat{z},\tilde{z}}) & \texttt{P}_i \neq \texttt{?} \\ z_i & \texttt{P}_i = \texttt{?} \end{cases}$,
(k_i) is the sequence of indices j for which $\texttt{P}_j = \texttt{?}$

Function arguments `Argument` and place holders `ArgumentPlaceholder` are translated in `FunctionCall` (59).

(60) `Argument ::= ExprSingle | ArgumentPlaceholder`
(61) `ArgumentPlaceholder ::= "?"`

B.3.9 Functions

Functions may be referenced by EQNames, as with function calls. Otherwise they are referenced via their name and arity or they are constructed inline.

(62) $[\![\texttt{FunctionItemExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\begin{cases} [\![\texttt{NamedFunctionRef(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{NamedFunctionRef} \\ [\![\texttt{InlineFunctionExpr(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{InlineFunctionExpr} \end{cases}$

Named function references `NamedFunctionRef` identify functions by its name and arity.

(63) $[\![\texttt{NamedFunctionRef(EQName(P)\#IntegerLiteral(P))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
x where $x \in \mathcal{C}onst$ is the function that matches $[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ and $[\![\texttt{IntegerLiteral(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(64) $[\![$`InlineFunctionExpr(function($EQName(x`$_1$`),...,$EQName(x`$_n$`))
{Expr(P)})`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\lambda z.$**match** z **with** $([\![$`EQName(x`$_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}}, \ldots, [\![$`EQName(x`$_n$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}})$ **in** $[\![$`Expr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$
$[\![$`InlineFunctionExpr(function($EQName(x`$_1$`) as SequenceType(P`$_1$`),...,`
`$EQName(x`$_n$`) as SequenceType(P`$_n$`))(as SequenceType(P'))`
`{Expr(P)})`$]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\lambda z.$**match** z **with** $(y_1, \ldots, y_n)$ **in**
let $[\![$`EQName(x`$_1$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = \mathit{convert\text{-}arg}^{\mathtt{P}_1}(y_1)$,
$\ldots$,
$[\![$`EQName(x`$_n$`)`$]\!]_{\bar{z},\hat{z},\tilde{z}} = \mathit{convert\text{-}arg}^{\mathtt{P}_n}(y_n)$
in $[\![$`Expr(P)`$]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.3.10 Simple Types

Simple types are atomic, list, and union types, into which nonterminal `SingleType` expands. Cast expressions require such simple types which however must directly translate nonterminal `SimpleTypeName` (88), such that nonterminal `SingleType` does not require a translation.

(65) `SingleType ::= SimpleTypeName "?"?`

TypeClaration are used when defining functions and function tests.

(66) `TypeDeclaration ::= "as" SequenceType`

B.4 Sequence Type Expressions

For matchings of some values v against sequence type expressions `P`, described in more detail in Section 4.5, the translation $[\![\mathtt{P}]\!]^v_{\bar{z},\hat{z},\tilde{z}}$ is further parametrized by some value v. The translation returns a boolean value depending on whether v matches a certain sequence type using the rules for sequence type matching[4].

B.4.1 Top level Sequence Type Matching

Sequence type matching is reduced to matching of item types. The empty sequence type matches the empty sequence.

(67) $[\![$`SequenceType(empty-sequence())`$]\!]^v_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
if $v = \mathit{nil}$ **then** *true* **else** *false*

An item type without an occurence indicator matches a sequence that contains exactly one item, if the item type matches that item.

$[\![$`SequenceType(ItemType(P))`$]\!]^v_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
match v **with** $x :: y$ **then**
if $y = \mathit{nil}$ **then** $[\![$`ItemType(P)`$]\!]^x_{\bar{z},\hat{z},\tilde{z}}$ **else** *false*
else *false*

Item types with an occurence indicator `?`, `*`, or `+` match a sequence, if the number of items in that sequence match the occurence indicator and the item types

[4] http://www.w3.org/TR/xpath-30/#id-sequencetype-matching

match the items in the sequence.

$[\![\texttt{SequenceType(ItemType(P)?)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} =_{\text{df}}$
  match v **with** $x :: y$ **then**
    if $y = nil$ **then** $[\![\texttt{ItemType(P)}]\!]^{x}_{\tilde{z},\hat{z},\tilde{z}}$ **else** *false*
  else *true*

$[\![\texttt{SequenceType(ItemType(P)*)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} =_{\text{df}}$
  let $rec = \lambda z.$**match** z **with** $x :: y$ **then**
      if $[\![\texttt{ItemType(P)}]\!]^{x}_{\tilde{z},\hat{z},\tilde{z}} = true$ **then** $rec(y)$ **else** *false*
    else *true*
  in $rec(v)$

$[\![\texttt{SequenceType(ItemType(P)+)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} =_{\text{df}}$
  let $rec = \lambda z.$**match** z **with** $x :: y$ **then**
      if $[\![\texttt{ItemType(P)}]\!]^{x}_{\tilde{z},\hat{z},\tilde{z}} = true$ **then** $rec(y)$ **else** *false*
    else *true*
  in if $v = nil$ **then** *false* **else** $rec(v)$

The OccurrenceIndicator is treated within the translation for sequence types `SequenceType`.

(68) `OccurrenceIndicator ::= "?" | "*" | "+"`

B.4.2 Item types

The matching against item types splits into the following tests. Notice that item type `item()` matches any item.

(69) $[\![\texttt{ItemType(P)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{KindTest(P)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{KindTest} \\ true & \texttt{P} = \texttt{item()} \\ [\![\texttt{FunctionTest(P)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{FunctionTest} \\ [\![\texttt{AtomicOrUnionType(P)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{AtomicOrUnionType} \\ [\![\texttt{ParenthesizedItemType(P)}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{ParenthesizedItemType} \end{cases}$$

B.4.3 Atomic and union types

Matching against atomic or union types reduces to testing whether the value is of the respective type.

(70) $[\![\texttt{AtomicOrUnionType(EQName(P))}]\!]^{v}_{\tilde{z},\hat{z},\tilde{z}} =_{\text{df}}$
  $type_{\text{TYPE}(v),\text{TYPE}([\![\texttt{EQName(P)}]\!]_{\tilde{z},\hat{z},\tilde{z}})}(v)$

B.4.4 Kind Tests

Kind tests match nodes of XML data trees against their type, name, and other properties.

(71) $[\![\texttt{KindTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{DocumentTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{DocumentTest} \\ [\![\texttt{ElementTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{ElementTest} \\ [\![\texttt{AttributeTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{AttributeTest} \\ [\![\texttt{SchemaElementTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{SchemaElementTest} \\ [\![\texttt{SchemaAttributeTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{SchemaAttributeTest} \\ [\![\texttt{PITest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{PITtest} \\ [\![\texttt{CommentTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{CommentTest} \\ [\![\texttt{TextTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{TextTest} \\ [\![\texttt{NamespaceNodeTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{NamespaceNodeTest} \\ [\![\texttt{AnyKindTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{AnyKindTest} \end{cases}$$

Type `node()` matches any node.

(72) $[\![\texttt{AnyKindTest(node())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$type_{\text{TYPE}(v),\mathbf{node}}(v)$

Type `document-node()` matches any document node.

(73) $[\![\texttt{DocumentTest(document-node())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$type_{\text{TYPE}(v),\mathbf{documentnode}}(v)$

The following two kind tests against document nodes, require the existence of exactly one element node child, that matches against a test for elements or a test for element with schema information, respectively.

$[\![\texttt{DocumentTest(document-node(ElementTest(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

if $type_{\text{TYPE}(v),\mathbf{documentnode}}(v)$ **then**
  match $set2list\{v' \mid \mathbf{at}(v, ch(is_{v'} \wedge labtype(v') = el))\}$ **with** $x :: y$
  then if $y = nil$ **then**
    $[\![\texttt{ElementTest(P)}]\!]^{v'}_{\bar{z},\hat{z},\tilde{z}}$
  else *false*
  else *false*
else *false*

$[\![\texttt{DocumentTest(document-node(SchemaElementTest(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

if $type_{\text{TYPE}(v),\mathbf{documentnode}}(v)$ **then**
  match $set2list\{v' \mid \mathbf{at}(v, ch(is_{v'} \wedge labtype(v') = el))\}$ **with** $x :: y$
  then if $y = nil$ **then**
    $[\![\texttt{SchemaElementTest(P)}]\!]^{v'}_{\bar{z},\hat{z},\tilde{z}}$
  else *false*
  else *false*
else *false*

Type `text()` matches any text node.

(74) $[\![\texttt{TextTest(text())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$type_{\text{TYPE}(v),\mathbf{textnode}}(v)$

Type `comment()` matches any comment node.

(75) $[\![\texttt{CommentTest(comment())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$type_{\text{TYPE}(v),\mathbf{commentnode}}(v)$

We do not support namespaces nodes and axes. However type `namespace-node()`

matches namespace nodes.

(76) $[\![\texttt{NamespaceNodeTest(namespace-node())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
raise *error*

Type `processing-instruction()` matches any processing-instruction node.

(77) $[\![\texttt{PITest(processing-instruction())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$type_{\text{TYPE}(v),\mathbf{pinode}}(v)$

Tests against processing-instruction nodes that specify an NCName, match any processing-instruction node whose target is equal to the NCName up to whitespace normalization.

$[\![\texttt{PITest(processing-instruction(NCName(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $x = [\![\texttt{NCName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
let $y = [\![\texttt{fn:normalize-space}]\!](x)$ **in**
if $\neg type_{\text{TYPE}(y),\mathbf{xs}:\mathbf{QName}}(y)$ **then raise** *error*(err:XPTY0004)
else $type_{\text{TYPE}(v),\mathbf{pinode}}(v) \wedge lab_{target}(v) = y$

If the XPath 1.0 compatibility mode were set to *true*, one would have to allow the specification of the processing-instruction target as string literal.

$[\![\texttt{PITest(processing-instruction(StringLiteral(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $x = [\![\texttt{StringLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
if $\neg type_{\text{TYPE}(x),\mathbf{xs}:\mathbf{QName}}(x)$ **then raise** *error*(err:XPTY0004)
else $type_{\text{TYPE}(v),\mathbf{pinode}}(v) \wedge lab_{target}(v) = x$

Type `attribute()` matches any attribute node.

(78) $[\![\texttt{AttributeTest(attribute())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$type_{\text{TYPE}(v),\mathbf{attribute}}(v)$

Attribute tests that specify an attribute name match attribute nodes with the respective node-name.

$[\![\texttt{AttributeTest(attribute(AttribNameOrWildcard(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$type_{\text{TYPE}(v),\mathbf{attribute}}(v) \wedge [\![\texttt{AttribNameOrWildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$

Attribute tests that besides an attribute name also specify a type, require in addition that the type annotation of the attribute node under consideration derives from the given type.

$[\![\texttt{AttributeTest(attribute(AttribNameOrWildcard(P),TypeName(P')))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$type_{\text{TYPE}(v),\mathbf{attribute}}(v) \wedge [\![\texttt{AttribNameOrWildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$
$\wedge derives\text{-}from(lab_{schema\text{-}type}(v), \text{TYPE}([\![\texttt{TypeName(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

(79) $[\![\texttt{AttribNameOrWildcard(AttributeName(P))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $z = [\![\texttt{AttributeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
$testEQName(v, z)$

$[\![\texttt{AttribNameOrWildcard(*)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
true

Schema attribute tests match attribute nodes, with a certain attribute declaration.

(80) $[\![\texttt{SchemaAttributeTest(schema-attribute(AttributeDeclaration(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$type_{\text{TYPE}(v),\mathbf{attribute}}(v) \wedge [\![\texttt{AttributeDeclaration(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$

The type $\text{TYPE}(s)$ for an attribute s denotes the schema type declared for s within the attribute declarations in $\mathcal{C}onst$.

(81) $[\![\texttt{AttributeDeclaration(AttributeName(P))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $z = [\![\texttt{AttributeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
$\mathit{testEQName}(v, z) \wedge \mathit{derives\text{-}from}(\mathit{lab}_{\mathit{schema\text{-}type}}(v), \text{TYPE}(z))$

Tests against element node, behave similar to tests for attribute nodes.

(82) $[\![\texttt{ElementTest(element())}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\mathit{type}_{\text{TYPE}(v),\textbf{elementnode}}(v)$

$[\![\texttt{ElementTest(element(ElementNameOrWildcard(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\mathit{type}_{\text{TYPE}(v),\textbf{elementnode}}(v) \wedge [\![\texttt{ElementNameOrWildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$

$[\![\texttt{ElementTest(element(ElementNameOrWildcard(P),TypeName(P')))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\mathit{type}_{\text{TYPE}(v),\textbf{elementnode}}(v) \wedge [\![\texttt{ElementNameOrWildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$
$\wedge \mathit{derives\text{-}from}(\mathit{lab}_{\mathit{schema\text{-}type}}(v), \text{TYPE}([\![\texttt{TypeName(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}))$
$\wedge \mathit{lab}_{\mathit{nilled}}(v) =$ "*false*"

$[\![\texttt{ElementTest(element(ElementNameOrWildcard(P),TypeName(P')?))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$\mathit{type}_{\text{TYPE}(v),\textbf{elementnode}}(v) \wedge [\![\texttt{ElementNameOrWildcard(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$
$\wedge \mathit{derives\text{-}from}(\mathit{lab}_{\mathit{schema\text{-}type}}(v), \text{TYPE}([\![\texttt{TypeName(P')}]\!]_{\bar{z},\hat{z},\tilde{z}}))$

(83) $[\![\texttt{ElementNameOrWildcard(ElementName(P))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
let $z = [\![\texttt{ElementName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$ **in**
$\mathit{testEQName}(v, z)$

$[\![\texttt{ElementNameOrWildcard(*)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
true

In order to correctly translate schema element tests `SchemaElementTest`, we require a formalism that allows to talk about the various properties of schema element declarations and substitution groups. We leave this translation open for future work.

(84) $[\![\texttt{SchemaElementTest(schema-element(ElementDeclaration(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
raise *error*

B.4.5 Node and Type Names

Not matching occurs for the following five nonterminals, who return simply names of elements and attributes, or names of types.

(85) $[\![\texttt{ElementDeclaration(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{ElementName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(86) $[\![\texttt{AttributeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(87) $[\![\texttt{ElementName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(88) $[\![\texttt{SimpleTypeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{TypeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

(89) $[\![\texttt{TypeName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}$

B.4.6 Function Tests

There are the following two types of function tests.

(90) $[\![\texttt{FunctionTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{AnyFunctionTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{AnyFunctionTest} \\ [\![\texttt{TypedFunctionTest(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{TypedFunctionTest} \end{cases}$$

Test `function(*)` matches any function.

(91) $[\![\texttt{AnyFunctionTest(function(*))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$type_{\text{TYPE}(v),\textbf{function}}(v)$

The specification for a typed function test verifies whether the sequence type of the signature of a given function is a subtype of the sequence type of an expected signature. This requires an extension in λXP that translates XPath functions to pairs of λXP function and XPath sequence type. Then the typed function test checks whether the corresponding XPath sequence type is a subtype of the expected sequence type. For checking of the subtype relation for sequence types of XPath we assume a function that does this.

(92)
```
TypedFunctionTest ::= "function"
  "(" (SequenceType ("," SequenceType)*)? ")" "as" SequenceType
```

Parenthesized item types match an item if and only it the item matches the item type within the parenthesis.

(93) $[\![\texttt{ParenthesizedItemType((ItemType(P)))}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$[\![\texttt{ItemType(P)}]\!]^{v}_{\bar{z},\hat{z},\tilde{z}}$

B.4.7 EQNames

EQNames are made up of either QNames or URIQualifiedNames.

(94) $[\![\texttt{EQName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$

$$\begin{cases} [\![\texttt{QName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{QName} \\ [\![\texttt{URIQualifiedName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} & \texttt{P} \in \texttt{URIQualifiedName} \end{cases}$$

B.5 Terminal Expressions

Terminal symbols `IntegerLieral`, `DecimalLiteral`, and `DoubleLiteral` translate to some numeric value of a respective type. Terminal symbols `StringLiteral`, `URIQualifiedName`, `Comment`, `QName`, `NCName`, and `Char` translate to a corresponding value. Note that non-numeric values which are no characters, are identified with a list of characters, such that they can be composed and decomposed.

(95) $[\![\texttt{IntegerLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(96) $[\![\texttt{DecimalLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(97) $[\![\texttt{DoubleLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(98) $[\![\texttt{StringLiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(99) $[\![\texttt{URIQualifiedName(BracedURILiteral(P) NCName(P'))}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}}$
$append([\![\texttt{BracedURILiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}}, [\![\texttt{NCName(P')}]\!]_{\bar{z},\hat{z},\tilde{z}})$

(100) $[\![\texttt{BracedURILiteral(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(101) $[\![\texttt{EscapeQuot(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(102) $[\![\texttt{EscapeApos(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(103) $[\![\texttt{Comment(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(104) $[\![\texttt{QName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(105) $[\![\texttt{NCName(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

(106) $[\![\texttt{Char(P)}]\!]_{\bar{z},\hat{z},\tilde{z}} =_{\text{df}} \texttt{P}$

Bibliography

[Alur 2004] Rajeev Alur and P. Madhusudan. *Visibly pushdown languages.* In Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004, pages 202–211, 2004. (Cited on pages 10 and 14.)

[Alur 2009] Rajeev Alur and P. Madhusudan. *Adding nesting structure to words.* J. ACM, vol. 56, no. 3, 2009. (Cited on pages 10, 24, 29, 30 and 121.)

[Arenas 2007] Marcelo Arenas, Pablo Barceló and Leonid Libkin. *Combining Temporal Logics for Querying XML Documents.* In Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings, pages 359–373, 2007. (Cited on page 6.)

[Arroyuelo 2015] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyen, Jouni Sirén and Niko Välimäki. *Fast in-memory XPath search using compressed indexes.* Softw., Pract. Exper., vol. 45, no. 3, pages 399–434, 2015. (Cited on page 9.)

[Atserias 2013] Albert Atserias, Martin Grohe and Dániel Marx. *Size Bounds and Query Plans for Relational Joins.* SIAM J. Comput., vol. 42, no. 4, pages 1737–1767, 2013. (Cited on page 2.)

[Barbieri 2010] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle and Michael Grossniklaus. *C-SPARQL: a Continuous Query Language for RDF Data Streams.* Int. J. Semantic Computing, vol. 4, no. 1, pages 3–25, 2010. (Cited on page 10.)

[Beame 2013] Paul Beame, Paraschos Koutris and Dan Suciu. *Communication steps for parallel query processing.* In Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013, pages 273–284, 2013. (Cited on page 2.)

[Beame 2014] Paul Beame, Paraschos Koutris and Dan Suciu. *Skew in parallel query processing.* In Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014, pages 212–223, 2014. (Cited on page 2.)

[Benedikt 2007] Michael Benedikt and Alan Jeffrey. *Efficient and Expressive Tree Filters.* In FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings, pages 461–472, 2007. (Cited on pages 10 and 11.)

[Benedikt 2008a] Michael Benedikt, Wenfei Fan and Floris Geerts. *XPath satisfiability in the presence of DTDs*. J. ACM, vol. 55, no. 2, 2008. (Cited on page 10.)

[Benedikt 2008b] Michael Benedikt, Alan Jeffrey and Ruy Ley-Wild. *Stream firewalling of xml constraints*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, pages 487–498, 2008. (Cited on pages 10 and 13.)

[Benedikt 2009] Michael Benedikt and Christoph Koch. *From XQuery to relational logics*. ACM Trans. Database Syst., vol. 34, no. 4, 2009. (Cited on page 4.)

[Benedikt 2015] Michael Benedikt, Gabriele Puppis and Huy Vu. *The complexity of higher-order queries*. Inf. Comput., vol. 244, pages 172–202, 2015. (Cited on page 14.)

[Berglund 2010] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. FernÃ¡ndez, Michael Kay, Jonathan Robie and Jérôme Siméon. *XML Path Language (XPath) 2.0, W3C recommendation*, 2010. http://www.w3.org/TR/xpath20/. (Cited on pages 5 and 6.)

[Beyer 2011] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan and Eugene J. Shekita. *Jaql: A Scripting Language for Large Scale Semistructured Data Analysis*. PVLDB, vol. 4, no. 12, pages 1272–1283, 2011. (Cited on page 2.)

[Bloom 2001] Gary Bloom and Christopher Lindblad. *MarkLogic*, 2001. http://www.marklogic.com/. (Cited on page 1.)

[Boag 2007] Scott Boag, Don Chamberlin, Mary F. FernÃ¡ndez, Daniela Florescu, Jonathan Robie and JÃ©rÃ´me SimÃ©on. *XQuery 1.0: An XML query language, W3C recommendation*, 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/. (Cited on page 4.)

[Bojańczyk 2011] Mikołaj Bojańczyk and Pawel Parys. *XPath evaluation in linear time*. J. ACM, vol. 58, no. 4, page 17, 2011. (Cited on page 8.)

[Bray 2008] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and FranÃ§ois Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. https://www.w3.org/TR/xml/. (Cited on pages 1 and 3.)

[Brenna 2009] Lars Brenna, Johannes Gehrke, Mingsheng Hong and Dag Johansen. *Distributed event stream processing with non-deterministic finite automata*. In Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009, 2009. (Cited on page 10.)

[Clark 1999a] James Clark. *XSL Transformations (XSLT) version 1.0, W3C recommendation*, 1999. https://www.w3.org/TR/xslt. (Cited on page 4.)

[Clark 1999b] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0, W3C recommendation*, 1999. hrefhttp://www.w3.org/TR/xpath/http://www.w3.org/TR/xpath/. (Cited on pages 4 and 5.)

[Comon 2007] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison and Marc Tommasi. *Tree Automata Techniques and Applications.*, 2007. http://www.grappa.univ-lille3.fr/tata. (Cited on page 4.)

[Crockford 2001] Douglas Crockford. *JSON (JavaScript Object Notation)*, 2001. http://json.org/, RFC 7159 and ECMA-404. (Cited on pages 2 and 23.)

[Debarbieux 2013] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian and Mohamed Zergaoui. *Early Nested Word Automata for XPath Query Answering on XML Streams*. In Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings, pages 292–305, 2013. (Cited on page 17.)

[Debarbieux 2015] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian and Mohamed Zergaoui. *Early nested word automata for XPath query answering on XML streams*. Theor. Comput. Sci., vol. 578, pages 100–125, 2015. (Cited on page 17.)

[Eich 2015] Brendan Eich. *JavaScript*, 2015. Standard ECMA-262. (Cited on page 2.)

[Engelfriet 2000] Joost Engelfriet and Sebastian Maneth. *Characterizing and Deciding MSO-Definability of Macro Tree Transductions*. In STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings, pages 542–554, 2000. (Cited on page 4.)

[Fernández 2007] Mary F. Fernández, Philippe Michiels, Jérôme Siméon and Michael Stark. *XQuery Streaming à la Carte*. In Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007, pages 256–265, 2007. (Cited on page 10.)

[Franceschet 2005] Massimo Franceschet. *XPathMark: An XPath Benchmark for the XMark Generated Data*. In Database and XML Technologies, Third International XML Database Symposium, XSym 2005, Trondheim, Norway, August 28-29, 2005, Proceedings, pages 129–143, 2005. revised version: http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html. (Cited on pages 12, 72, 143 and 253.)

[Friedgut 2004] Ehud Friedgut. *Hypergraphs, Entropy, and Inequalities*. The American Mathematical Monthly, vol. 111, no. 9, pages 749–760, 2004. (Cited on page 2.)

[Frisch 2004] Alain Frisch. *Regular Tree Language Recognition with Static Information*. In Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France, pages 661–674, 2004. (Cited on page 154.)

[Gauwin 2009a] Olivier Gauwin. *Streaming Tree Automata and XPath*. PhD thesis, Université Lille 1, 2009. (Cited on page 128.)

[Gauwin 2009b] Olivier Gauwin, Joachim Niehren and Sophie Tison. *Earliest Query Answering for Deterministic Nested Word Automata*. In Fundamentals of Computation Theory, 17th International Symposium, FCT 2009, Wroclaw, Poland, September 2-4, 2009. Proceedings, pages 121–132, 2009. (Cited on pages 10, 14, 116, 121 and 127.)

[Gauwin 2011a] Olivier Gauwin and Joachim Niehren. *Streamable Fragments of Forward XPath*. In Implementation and Application of Automata - 16th International Conference, CIAA 2011, Blois, France, July 13-16, 2011. Proceedings, pages 3–15, 2011. (Cited on pages 10, 117, 121 and 148.)

[Gauwin 2011b] Olivier Gauwin, Joachim Niehren and Sophie Tison. *Queries on Xml streams with bounded delay and concurrency*. Inf. Comput., vol. 209, no. 3, pages 409–442, 2011. (Cited on page 11.)

[Goldfarb 1986] Charles Goldfarb, Edward Mosher and Raymond Lorie. *The Standard Generalized Markup Language (SGML)*, 1986. ISO 8879:1986. (Cited on page 1.)

[Gottlob 2002] Georg Gottlob and Christoph Koch. *Monadic Queries over Tree-Structured Data*. In 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings, pages 189–202, 2002. (Cited on pages 4 and 5.)

[Gottlob 2003] Georg Gottlob, Christoph Koch and Reinhard Pichler. *The complexity of XPath query evaluation*. In Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA, pages 179–190, 2003. (Cited on page 5.)

[Gottlob 2005] Georg Gottlob, Christoph Koch and Reinhard Pichler. *Efficient algorithms for processing XPath queries*. ACM Trans. Database Syst., vol. 30, no. 2, pages 444–491, 2005. (Cited on page 8.)

[Green 2004] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka and Dan Suciu. *Processing XML streams with deterministic automata and stream indexes*. ACM Trans. Database Syst., vol. 29, no. 4, pages 752–788, 2004. (Cited on page 144.)

[Groz 2012] Benoit Groz. *XML Security Views: Queries, Updates, and Schemas*. PhD thesis, Université Lille 1, 2012. (Cited on page 8.)

[Grün 1983] C. Grün. *BaseX. The XML Database*, 1983. http://basex.org/. (Cited on page 1.)

[Grust 2007] Torsten Grust, Jan Rittinger and Jens Teubner. *eXrQuy: Order Indifference in XQuery*. In Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007, pages 226–235, 2007. (Cited on page 69.)

[Gupta 2003] Ashish Kumar Gupta and Dan Suciu. *Stream Processing of XPath Queries with Predicates*. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, pages 419–430, 2003. (Cited on page 10.)

[Hickson 2014] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor and Silvia Pfeiffer. *HTML5*, 2014. http://www.w3.org/TR/html5/. (Cited on page 1.)

[Jelliffe 2006] Rick Jelliffe. *Schematron*, 2006. ISO/IEC 19757-3. (Cited on page 3.)

[Kay 2004] Michael Kay. *The saxon XSLT and XQuery processor*, 2004. https://www.saxonica.com. (Cited on pages 9, 153 and 166.)

[Kay 2010] Michael Kay. *A streaming XSLT processor*. In Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, volume 5, MontrÃ©al, Canada, August 3 - 6, 2010, 2010. (Cited on pages 10, 11 and 144.)

[Kay 2014] Michael Kay. *XPath and XQuery Functions and Operators 3.0, W3C recommendation*, 2014. https://www.w3.org/TR/xpath-functions-30/. (Cited on pages 13 and 63.)

[Kupferman 2001] Orna Kupferman and Moshe Y. Vardi. *Model Checking of Safety Properties*. Formal Methods in System Design, vol. 19, no. 3, pages 291–314, 2001. (Cited on page 10.)

[Labath 2015] Pavel Labath and Joachim Niehren. *A Uniform Programming Language for Implementing XML Standards*. In SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings, pages 543–554, 2015. (Cited on page 252.)

[Ley 2009] Clemens Ley and Michael Benedikt. *How big must complete XML query languages be?* In Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings, pages 183–200, 2009. (Cited on pages 143 and 145.)

[Lick 2013] Anthony Lick and Joachim Niehren. *Early = Earliest?* Rapport technique, INRIA Lille, 2013. http://hal.inria.fr/hal-00873742. (Cited on pages 15, 117, 126 and 148.)

[Madhusudan 2009] P. Madhusudan and Mahesh Viswanathan. *Query Automata for Nested Words.* In Mathematical Foundations of Computer Science 2009, 34th International Symposium, MFCS 2009, Novy Smokovec, High Tatras, Slovakia, August 24-28, 2009. Proceedings, pages 561–573, 2009. (Cited on pages 10, 14, 116, 118 and 127.)

[Maneth 2005] Sebastian Maneth, Alexandru Berlea, Thomas Perst and Helmut Seidl. *XML type checking with macro tree transducers.* In Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA, pages 283–294, 2005. (Cited on page 4.)

[Maneth 2010] Sebastian Maneth and Kim Nguyen. *XPath Whole Query Optimization.* PVLDB, vol. 3, no. 1, pages 882–893, 2010. (Cited on pages 9, 153 and 154.)

[Marian 2003] Amélie Marian and Jérôme Siméon. *Projecting XML Documents.* In VLDB, pages 213–224, 2003. (Cited on pages 9 and 153.)

[Martens 2006a] Wim Martens. *Static Analysis of Xml Transformation and Schema Languages.* PhD thesis, Universiteit Antwerpen (Belgium), 2006. (Cited on page 3.)

[Martens 2006b] Wim Martens, Frank Neven, Thomas Schwentick and Geert Jan Bex. *Expressiveness and complexity of XML Schema.* ACM Trans. Database Syst., vol. 31, no. 3, pages 770–813, 2006. (Cited on page 3.)

[Marx 2004a] Maarten Marx. *XPath with Conditional Axis Relations.* In Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings, pages 477–494, 2004. (Cited on page 6.)

[Marx 2004b] Maarten Marx and Maarten de Rijke. *Semantic Characterization of Navigational XPath.* In First Twente Data Management Workshop (TDM 2004) on XML Databases and Information Retrieval, Enschede, The Netherlands, June 21, 2004, pages 73–79, 2004. (Cited on page 6.)

[Meier 2000] Wolfgang Meier. *eXist*, 2000. http://exist-db.org/. (Cited on page 1.)

[Mozafari 2012] Barzan Mozafari, Kai Zeng and Carlo Zaniolo. *High-performance complex event processing over XML streams*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, pages 253–264, 2012. (Cited on pages 10, 11, 13 and 144.)

[Ngo 2013] Hung Q. Ngo, Christopher Ré and Atri Rudra. *Skew strikes back: new developments in the theory of join algorithms*. SIGMOD Record, vol. 42, no. 4, pages 5–16, 2013. (Cited on page 2.)

[Olteanu 2007] Dan Olteanu. *SPEX: Streamed and Progressive Evaluation of XPath*. IEEE Trans. Knowl. Data Eng., vol. 19, no. 7, pages 934–949, 2007. (Cited on pages 10, 11, 12, 116 and 144.)

[Onizuka 2010] Makoto Onizuka. *Processing XPath queries with forward and downward axes over XML streams*. In EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings, pages 27–38, 2010. (Cited on page 144.)

[Parys 2011] Pawel Parys. *Application of automata theory to processing of XML documents*. PhD thesis, Warsaw University, 2011. (Cited on pages 8 and 9.)

[Pemberton 2002] Steven Pemberton and ... *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, 2002. http://www.w3.org/TR/xhtml1/. (Cited on page 1.)

[Peng 2005] Feng Peng and Sudarshan S. Chawathe. *XSQ: A streaming XPath engine*. ACM Trans. Database Syst., vol. 30, no. 2, pages 577–623, 2005. (Cited on page 144.)

[Phuoc 2011] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira and Manfred Hauswirth. *A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*. In The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, pages 370–388, 2011. (Cited on page 10.)

[Robie 2014a] Jonathan Robie, Don Chamberlin, Michael Dyck and John Snelson. *XML Path Language (XPath) 3.0, W3C recommendation*, 2014. http://www.w3.org/TR/xpath-30/. (Cited on page 5.)

[Robie 2014b] Jonathan Robie, Don Chamberlin, Michael Dyck and John Snelson. *XPath 3.0 Grammar, W3C recommendation*, 2014. https://www.w3.org/TR/xpath-30/#nt-bnf. (Cited on pages 13, 26, 41, 42, 47 and 102.)

[Robie 2014c] Jonathan Robie, Don Chamberlin, Michael Dyck and John Snelson. *XQuery 3.0: An XML Query Language, W3C recommendation*, 2014. https://www.w3.org/TR/xquery-30/. (Cited on page 5.)

[Robinson 2015] Ian Robinson, Jim Webber and Emil Eifrem. Graph databases. new opportunities for connected data. O'Reilly Media, 2015. (Cited on page 2.)

[Schmidt 2007] Michael Schmidt, Stefanie Scherzinger and Christoph Koch. *Combined Static and Dynamic Analysis for Effective Buffer Minimization in Streaming XQuery Evaluation*. In Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007, pages 236–245, 2007. (Cited on pages 10, 11, 12 and 144.)

[Sebastian 2016] Tom Sebastian and Joachim Niehren. *Projection for Nested Word Automata Speeds up XPath Evaluation on XML Streams*. In SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings, pages 602–614, 2016. (Cited on page 17.)

[ten Cate 2007] Balder ten Cate and Maarten Marx. *Navigational XPath: calculus and algebra*. SIGMOD Record, vol. 36, no. 2, pages 19–26, 2007. (Cited on page 8.)

[Thatcher 1968] James W. Thatcher and Jesse B. Wright. *Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic*. Mathematical Systems Theory, vol. 2, no. 1, pages 57–81, 1968. (Cited on page 4.)

[Thompson 2004] Henry S. Thompson, David Beech, Murray Maloney and Noah Mendelsohn. *XML Schema Part 1: Structures Second Edition*, 2004. http://www.w3.org/TR/xmlschema-1/. (Cited on page 3.)

[van der Vlist 2003] Eric van der Vlist. Relax ng. O'Reilly Media, 2003. (Cited on page 3.)

[Walsh 2010a] Norman Walsh. Docbook: The definitive guide. O'Reilly Media, 2010. (Cited on page 1.)

[Walsh 2010b] Norman Walsh, Alex Milowski and Henry S. Thompson. *XProc: An XML Pipeline Language, W3C Recommendation*, 2010. https://www.w3.org/TR/xproc/. (Cited on page 4.)

[Wu 2006] Eugene Wu, Yanlei Diao and Shariq Rizvi. *High-performance complex event processing over streams*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006, pages 407–418, 2006. (Cited on page 10.)